2

Final Report

Study of the Use of Ada in Trusted Computing Bases (TCBs) to be Certified at, or Below, the B3 Level

Prepared for:

National Computer Security Center
9800 Savage Road
Fort Meade, MD 20755

Prepared by:

Ada Applications and
Software Technology Group

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

April 1989

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE April 1989	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Final Report: Study of the Use of Ada in Trusted Computer Bases (TCBs) to be Certified at, or Below, the B3 Level	5. FUNDING NUMBERS
6. AUTHOR(S)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ada Applications and Software Technology Group IIT Research Institute 4600 Forbes Blvd Lanham, MD 20706	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Computer Security Center 9800 Savage Road Fort Meade, MD 20755	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	15. NUMBER OF PAGES 277
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5500

Standard Form 298, (Rev. 2-89
Prescribed by ANSI Std. 239-18
298-01

TABLE OF CONTENTS

1.0 INTRODUCTION

This is the final report for the "Study of the Use of Ada in Trusted Computing Bases (TCBs) to be certified at, or below, the B3 Level." The objective of the study was to produce guidelines for developing Ada software for TCBs. This objective was addressed in a three-part process:

1) Mapping the Trusted Computer System Evaluation Criteria (TCSEC) to the software development process,

2) Identifying benefits of and potential deterrents to using Ada in the software development process of TCB systems, AND

3) Producing the guidelines. (KR)

The mapping of the TCSEC to the software development process was done to identify precisely where and how application of the TCSEC affected the software development process. This task is described in Section 2.0. The mapping is contained in Appendix A.

The identification of benefits of and potential deterrents to using Ada in TCBs was based on knowing where and how TCSEC criteria affected software development. These benefits and deterrents form the rationale for the guidelines. The rationale is described in Section 3.0, and the listing of benefits and potential deterrents is in Appendix B.

The guidelines produced are based on the rationale. The use of the guidelines is discussed in Section 4.0, and the guidelines are presented in Appendix C. The guidelines are intended to supplement developer-selected guidelines for implementing a TCB as well as general-purpose Ada programming guidelines.

Examples of Ada code are presented in Appendices B and C. In Appendix B, the code examples typically illustrate why the use of a specific Ada construct must be limited in a particular fashion if that construct is to be used in a TCB. The examples in Appendix C illustrate how particular constructs can be used in the development of a TCB.

Key terms that are used in this document are listed in Section 5.0, and the bibliography is in Section 6.0. Because each appendix is intended to be a stand-alone document, each also contains a key terms section and a bibliography.

This report does not imply that software alone is sufficient for ensuring security in a TCB. As discussed in the papers "Secure Computing: The Secure Ada Target Approach," "LOCKing Computers Securely," and "LOCK/ix: On Implementing Unix on the LOCK TCB," the security of a system cannot be insured with software alone. Hardware is fundamentally important to TCB system security. This report does not discuss hardware aspects of TCB systems. If the reader wishes to investigate the hardware issues, he should consult the references cited above. An abstract for each is presented in Appendix D.

This study is somewhat similar to reports written on the development of the Army Secure Operating System (ASOS).

The analysis of Ada for security performed by TRW in its development of the Army Secure Operating System (ASOS) has a different objective and viewpoint than that of this study on the use of Ada in trusted computing bases (TCBs). The TRW analysis is documented in its final report, Multilevel Secure Operating System Final Development Specification Rationale for the Army Secure Operating System (ASOS). The introduction of Chapter 5, "Analysis of Ada for Security," states the following:

> "This study partitions the problems concerning Ada and security into the following three categories:"
>
> a. "Problems that arise from the A1 requirement . . . to do code correspondence, i.e., to show the correspondence between the formal top level specifications and the source code that implements these specifications."
>
> b. "Problems that arise from Ada's need for an elaborate runtime support library."
>
> c. "Problems that arise from the (beyond A1) considerations of Ada code verifications."

In contrast this study attempts to identify Ada software development guidelines for creating trusted Ada code for TCBs. It does not address TCBs above the B3 class. Therefore it did not attempt to identify benefits, deterrents, and guidelines with the criteria of Ada code correspondence with formal specifications and formal Ada code verification (i.e., items a and c above). Nor does this study use the criterion of minimizing the size of the runtime support library in the development of the security kernel (i.e., the reference monitor). Unlike the ASOS this study makes no distinction between Ada constructs that are allowed inside and outside of a TCB's security kernel.

Despite the differences between the two studies, they are not inconsistent with each other. Rather they complement each other. Many similar guidelines appear in the two studies. Pertinent ASOS guidelines have been borrowed and incorporated into this report and are indicated as such.

2.0 RELATING THE TCSEC TO THE SOFTWARE DEVELOPMENT PROCESS

2.1 Background

The first step in developing Ada programming guidelines for software to which the TCSEC is to be applied is to map the TCSEC to the software development process. This is to assure that the relationship between elements of the software development process and the specific TCSEC criteria are understood. Because this mapping is language independent, Ada is not specifically mentioned in the mapping. The details of the mapping are contained in Appendix A, "A Mapping from the Trusted Computer System Evaluation Criteria (TCSEC) to the Software Development Process."

The intent of this process is to detail what must be accomplished at each stage of a software development to optimize the certification of a system using the Department of Defense TCSEC. This optimization is from two perspectives: one is to ensure that the certification process meets the objective of understanding what the software product will, and will not, do; the other is to reduce the effort required to perform the certification process.

The intent of this section is to enable the reader to better use the mapping in Appendix A. The first area discussed in this section is the life cycle model used for the purposes of this report. The next section contains general comments on software development that are germane to developing software that is to be certified using TCSEC.

2.2 Life Cycle Description

Several models of the software development life cycle exist. These include the waterfall model, articulated by DoD-STD-2167A Defense System Software Development, and the incremental development model represented by Dr. Barry Boehm's spiral development model [Boehm 1988]. The primary difference between these models is that the waterfall model assumes that one phase of development is completed prior to commencement of the next phase, whereas incremental development iterates between phases and leads to partial system development with increments to the system being preplanned.

Rather than focusing on the distinctions between these two, or other, development models, this document assumes that systems pass through specific phases during their development and operational life cycles. Regardless of whether these phases are entered only once during a development or are entered iteratively, the phases are adequately generic to be used here. The five phases used are the following:

1. Requirements

2. Design

3. Coding

4. Testing

5. Support

These phases are used to structure the guidance as to what is to be accomplished during software development to optimize the certification process. A checklist of specific accomplishments is provided in Appendix A for each of these five phases. The checklists are developed to provide maximum latitude on how each item is to be implemented. This is to ensure that no development methodology is either the required or the implied standard. Following each checklist is a textual explanation of the items on the list.

2.3 General Comments on the Software Development Process

This research effort focuses on software development, which is an element of system development. Although the results of this effort will be applicable to system developers, these results are specifically targeted to software developers. Therefore, terms that can be applied to system development or software development should be interpreted from the software development perspective.

The software development process for certified systems will not vary significantly from that for conventional systems. The process will be accomplished through the completion of various software development phases including the requirements phase, the design phase, the coding phase, the testing phase, and the support phase. Each of these phases, as detailed in Section 2.2, progresses as in the development of conventional systems with some additional considerations, such as the use of defensive programming and defensive testing approaches. These additional considerations will aid the software developer by providing assistance either to assure that the goals of the TCSEC are met, to implement the TCSEC criteria, or to enhance an existing system without requiring excessive effort in re-certification. In particular, the design phase, the coding phase, and the testing phase each require additional consideration during the software development of a TCB.

During the design phase of software development, the software developer must use a design methodology that is compatible with the requirements analysis. By ensuring this compatibility, the software developer can ensure that there will be traceability between the requirements and the design. This traceability will demonstrate the completeness of the system software, which, in turn, guarantees that all software requirements are addressed in the design.

In addition, during the design phase of the software development, the developer needs to identify the various protection mechanisms that will be implemented during development. Errors and omissions are the leading causes of security breaches, and these can be addressed by design reviews and code walkthroughs, for example. Whatever protection mechanisms are to be implemented, the implementation details must be determined during the design phase, and an appropriate design must be established.

The coding phase of the software development of TCBs, differs from that of conventional systems, primarily, in that the programmer must be particularly attentive to the use of coding standards that promote and do not compromise the code's integrity. When coding a conventional system the programmer attempts to satisfy the requirements as stated in the software requirements documentation. These requirements identify both the data to be handled by the system and the operations required to handle the data. However, although a TCB's software requirements document contains the same types of information as that for a conventional system, the developer of a TCB must also code in a manner so that the operations are implemented so as not to compromise the data for which the TCB is responsible. For example, if users of types X, Y, and Z should be allowed to perform operation A, then when coding operation A, the software developer needs to create the software to check the type of the user to ensure not only that it is one of type X, Y, or Z, but also that it is not one of the other types of users.

Dishonest and disgruntled employees are major threats to developing secure systems. Management must make every effort to ensure that the programmers are adequately screened prior to their involvement with a trusted system. The extent of the required background checks of the employees is a function of the sensitivity of the system. The more sensitive the system, the more thorough the background check must be. Also, monitoring of programmer attitudes during development is necessary to identify potential personnel problems early. Although various techniques exist to ensure the consistency between the design and code of a system and to ensure the necessity and function of sections of code, these techniques are far from being perfected. Because there is no technical mechanism that provides assurance that malicious code nas not been introduced, background checks and monitoring programmer attitudes are necessary complements to software development practices.

Finally, during the testing phase for certified systems, the testing process goes beyond that for traditional software. In addition to testing to ensure that the software supports appropriate users and uses, the software must deny access to inappropriate users and not support inappropriate uses. Safeguards against accidental access to system data, against disclosing information about the system's structure, and against providing information about system users must be tested. The testing must assure that accidental or intentional system misuse does not compromise the security of the system, its data, or its users. Knowing that the testing process for certified systems goes beyond that for conventional systems, the design of certified systems must support these testing needs. An initial step is to inform the system designers of the testing requirements of the system so that the design supports the testing of safeguards. In addition, test hooks may be designed into the system, where the purpose of the hooks is to support the testing process.

The intent of this project is to be unbiased toward the various Ada design methodologies; therefore, Ada design issues raised in the reports should apply to all Ada design methodologies. Items included are various Ada facilities that aid good software development, such as the use of packages, subprograms, data types (especially private types), etc., for information hiding, modularity, and data abstraction.

2.4 Format of the Mapping

The format of Appendix A follows Part I of the TCSEC. Each TCB class, from C1 through B3, is described. The four major subheadings of TCBs, Security Policy, Accountability, Assurance, and Documentation, are then presented, with a synopsis of the certification criteria for each. Each subheading contains a checklist of activities for each of the five phases of the software development process.

3.0 RATIONALE FOR THE PROGRAMMING GUIDELINES

3.1 Background

The first step in developing Ada programming guidelines for software to which the TCSEC is to be applied was to map the TCSEC to the software development process. The second was to develop the rationale on which the guidelines are to be based. The rationale for the guidelines are the benefits and potential deterrents of the Ada programming language relative to the needs of the TCSEC.

One intent of the rationale is to identify benefits of using Ada in the software development of TCBs. These benefits include Ada's assets in the application of sound software engineering principles, such as data abstraction, information hiding, modularity, and localization. Also included among the benefits are such Ada constructs as strong data typing, packages, subprograms, and tasks.

Another intent of the rationale is to identify and categorize potential deterrents of using Ada in the software development of TCBs. These include shortcomings inherent to programming languages in general; shortcomings unique to the Ada language; and benefits of using tools in the development of Ada language software.

3.2 Format of Rationale

The rationale for the guidelines are presented in Appendix B, Benefits of and Potential Deterrents to Using Ada in the Software Development Process of Trusted Computing Bases. The rationale are presented from two perspectives. First, the Ada language is considered by focusing on the following issues: (1) Benefits of using Ada in developing TCB systems, (2) Potential deterrents to using Ada in developing TCB systems, (3) Shortcomings inherent to programming languages in general in developing TCB systems, (4) Shortcomings unique to the Ada language in developing TCB systems, (5) Benefits of using tools for developing Ada software for TCB systems. The rationale also presents these issues in the context of a mapping of Ada usage to generalized TCB criteria. In particular, Class B3 is used as a template for the generalized TCB criteria. Ada constructs and features are identified that may be used to implement TCB functions and features.

3.3 Conclusions

Because Ada was designed with features and constructs that promote recognized sound software engineering principles more than other current high order languages (HOLs), it is well suited as the implementation language of TCBs. Although Ada offers various specific benefits, the potential deterrents of using Ada must be recognized and addressed. Although these potential problems were identified with using various Ada features, this is not meant to imply that any of the features should not be used. Rather, the security of the TCB must not be compromised, as discussed in Appendix B, when any of the constructs and features are used in the implementation of the TCB.

4.0 USE OF THE GUIDELINES

4.1 Objective, Format, and Scope of the Guidelines

The guidelines presented in Appendix C, Programming Guidelines for Using Ada in the Software Development of TCBs, have a very specific focus. That focus is to assist developers who are both experienced with the TCSEC and knowledgeable of Ada in merging these two technologies. The objective of the guidelines is twofold:

1. Provide direction on the use of particular Ada constructs

2. Provide recommendations on how to implement specific TCSEC criteria with Ada.

Because of this dual objective, the guidelines are presented using two frameworks. The first presentation is a listing of the chapters and sections of The Reference Manual for the Ada Programming Language (IRM) [ANSI/MIL-STD-1815A-1983], and descriptions and guidelines, where appropriate, on the use of sections for TCSEC software. The second presentation is a listing of the criteria of the TCSEC, using B3 criteria as a template, with guidelines on how to implement the criteria using Ada.

The scope of these guidelines is specific to the TCSEC and to the Ada programming language. The guidelines would be best used, therefore, in conjunction with internal company standards for software engineering of TCBs and for software engineering with Ada. Modularization, for example, is a key software engineering principle that should be incorporated into a system design. Specific TCSEC criteria whose implementation is facilitated by modularization are noted. The Ada structures that could be used for modularization to support the specific criteria are listed.

These guidelines are, therefore, intended to complement other standards, not to stand alone. Specifically, developer-selected guidelines for implementing a TCB as well as general-purpose Ada programming guidelines should be supplemented with, not supplanted by, these guidelines.

4.2 Limitations of the Language Constructs

The Ada language is rich and powerful in its programming constructs. With this breadth of constructs comes the potential to write software that is difficult to understand and control. Concurrency, for example, is necessary in several applications. Concurrent software, however, is very difficult to test, understand and predict. The Ada construct for implementing concurrent processes is the Ada task. TCSEC software must be testable, understandable, and predictable. Also, many secure applications require concurrent processes. The guidelines cannot realistically recommend that tasks, for example, never be used. Instead, they list where tasks are appropriate and they discuss how tasks are to be implemented to minimize the potential negative effects of using tasks. The guidelines do not recommend prohibiting the use of any construct but rather, in some cases, recommend limits on how constructs are to be used.

Constructs other than tasks have both advantages and disadvantages. A typical tradeoff is between memory usage efficiency and execution time efficiency. Dynamic memory structures, for example, are typically efficient from the memory usage perspective but inefficient from the execution timing perspective. The guidelines state the tradeoffs involved in using various constructs to facilitate application-specific decisions.

5.0 KEY TERMS

Several key words appear throughout the text of this document. These words have specific meanings within the context of certified systems, and their definitions are presented here.

These definitions are taken directly from the TCSEC:

Access - A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

Audit Trail - A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions.

Covert Channel - A communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Data - Information with a specific physical representation.

Discretionary Access Control - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Mandatory Access Control - A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

Object - A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc.

Sensitivity Label - A piece of information that represents the security level of an object and that describes the sensitivity (e.g., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions.

Subject - An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

Trusted Computing Base (TCB) - The totality of protection mechanisms within a computer system — including hardware firmware, and software — the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Additional Terms

These terms are included because they appear frequently in the following text.

Pragma - A compiler directive. That is, it "is used to convey information to the compiler." According to the Ada language reference manual [ANSI/MIL-STD-1815A-1983], the predefined pragmas (Refer to Annex B in this manual for descriptions) "must be supported by every implementation. In addition, an implementation may provide implementation-defined pragmas, which must then be described in Appendix F", i.e., the appendix on implementation-dependent characteristics that the Ada compiler vendor must provide in his Ada language reference manual.

Security - The protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure. Security also pertains to personnel, data, communications, and the physical protection of computer installations [IEEE 1983]. Specifically, for the purposes of this report, security is defined by the criteria in the TCSEC, i.e., a given security problem corresponds with a specific TCSEC criteria.

6.0 BIBLIOGRAPHY

Abrams, Marshall D., Podell, Harold J., 1987. Tutorial Computer and Network Security. Washington, D. C.: IEEE Computer Science Press.

Abrams, Marshall D., Podell, Harold J., 1988. Recent Developments in Network Security. 2906 Covington Road, Silver Spring, MD, 20910.: Computer Educators Inc.

Anderson, Eric R. "Ada's Suitability for Trusted Computer Systems" from Proceedings of the Symposium on Security and Privacy, Oakland, California, 22-24 April, 1985.

Baker, T. P. 13 July 1988. Issues Involved in Developing Real-Time Ada Systems. Department of Computer Science, Florida State University, Tallahasse, FL: for U. S. Army HQ, COMM/ADP.

Boebert, W. E., Kain, R. Y., and Young, W. D., July 1985. "Secure Computing: The Secure Ada Target Approach." Scientific Honeyweller, Vol. 6, No. 2.

Booch, Grady. 1987A. Software Components with Ada. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Booch, Grady. 1987B. Software Engineering with Ada. 2nd ed. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Brill, Alan E., 1983. Building Controls Into Structured Systems. New York, N. Y.: YOURDON Press Inc.

Buhr, R. J. A., 1984. System Design with Ada. Englewood Cliffs, N. J.: Prentice-Hall.

Cherry, George W., 1984. Parallel Programming in ANSI Standard Ada. Reston, Virginia: Reston Publishing Company, Inc.

Feldman, Michael B. 1985. Data Structures with Ada. Reston, Virginia: Reston Publishing Company, Inc.

Final Evaluation Report of SCOMP 23 September 1985. Secure Communications Processor STOP Release 2.1.

Freeman, Peter. 1987. Tutorial: Software Reusability. Washington, D. C.: IEEE Computer Science Press.

Gasser, Morrie 1988. Building a Secure Computer System. New Y ' N. Y.: Van Nostrand Reinhold Company, Inc.

Gehani, Narain. 1984. Ada Concurrent Programming. Englewooa Cliffs, N. J.: Prentice-Hall Inc.

Gilpin, Geoff. 1986. Ada: A Guided Tour and Tutorial. New York, N. Y.: Prentice Hall Press.

Goodenough, John B., "Exception Handling: Issues and a Proposed Notation", Communications of the ACM, 18(12):683-696, December 1975.

Hadley, Sara, Hellwig, Frank G. of the National Security Agency, and Rowe, Kenneth, CDR Vaurio, David of the National Computer Security Center. 1988. "A Secure SDS Software Library," Proceedings, 11th National Computer Security Conference, Baltimore, MD, October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.

IEEE Standard Glossary of Software Engineering Terminology. 18 February 1983. (IEEE Std 729-1983).

Luckham, David C., von Henke, Friendrich W., Krieg-Brueckner, Bernd, Owe, Olaf, "ANNA-A Language for Annotating Ada Programs, Preliminary Reference Manual", Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

Mungle, Jerry. 1988. Developing Ada Systems. Technology Training Corporation's seminar.

National Computer Security Center. 1985. Department of Defense Trusted Computer System Evaluation Criteria. (DOD 5200.28-STD)

National Computer Security Center. 1987. Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria.

Nissen, John and Wallis, Peter. 1984. Portability and Style in Ada. Cambridge, Great Britain: Cambridge University Press.

Odyssey Research Associates, Inc., Toward Ada Verification, Preliminary Report (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, March 25, 1985.

Reference Manual for the Ada Programming Language. 1983. ANSI/MIL-STD-1815A-1983, 17 February 1983.

Ross, D. T., Goodenough, J. B., and Irvine, C. A., 1975. "Software Engineering: Process, Principles, and Goals," Computer.

Saydjari, O. S., Beckman, J. M., and Leaman, J. R. 1987. "LOCKing Computers Securely," Proceedings, 10th National Computer Security Conference, Baltimore, MD, September 21-24, 1987, National Bureau of Standards/National Computer Security Center.

Shaffer, Mark of Honeywell, Computing Technology Center, and Walsh, Geoff of R & D Associates Secure. 1988. "LOCK/ix: On Implementing Unix on the LOCK TCB," Proceedings, 11th National Computer Security Conference, Baltimore, MD, October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.

Tracz, Will. 1988. Tutorial: Software Reuse: Emerging Technology. Washington, D. C.: IEEE Computer Science Press.

Tripathi, Anand R., Young, William D., Good, Donald I., "A Preliminary Evaluation of Verifiability in Ada", Proceedings of the ACM National Conference, Nashville, TN, October 1980.

Trusted Computer System Security Requirements Guide for DoD Applications. 1 September 1987.

APPENDIX A

A Mapping
from the
Trusted Computer System Evaluation Criteria (TCSEC)
to the
Software Development Process

Prepared for:

National Computer Security Center
9800 Savage Road
Fort Meade, MD 20755

Prepared by:

Ada Applications and
Software Technology Group

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

April 1989

APPENDIX A
TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Background

The intent of this appendix is to detail what must be accomplished at each stage of a software development to optimize the certification of a system using the Department of Defense TCSEC. This optimization is from two perspectives: one is to ensure that the certification process meets the objective of understanding what the software product will, and will not, do; the other is to reduce the effort required to perform the certification process. Because this mapping is language independent, Ada is not mentioned in the body of the this Appendix.

This appendix is meant to stand alone; however, a strong familiarity with the TCSEC is required to use the document.

1.2 Life Cycle Description

Several models of the software development life cycle exist. These include the waterfall model, articulated by DoD-STD-2167A Defense System Software Development, and the incremental development model represented by Dr. Barry Boehm's spiral development model [Boehm 1988]. The primary difference between these models is that the waterfall model assumes that one phase of development is completed prior to commencement of the next phase, whereas incremental development iterates between phases and leads to partial system development with increments to the system being preplanned.

Rather than focusing on the distinctions between these two, or other, development models, this document assumes that systems pass through specific phases during their development and operational life cycles. Regardless of whether these phases are entered only once during a development or are entered iteratively, the phases are adequately generic to be used to structure this report. The five phases used here are the following:

1. Requirements
2. Design
3. Coding
4. Testing
5. Support

These phases are used to structure the guidance as to what is to be accomplished during software development to optimize the certification process. A checklist of specific accomplishments is provided for each of these five phases. The checklists are developed to provide maximum latitude on how each item is to be implemented. This is to ensure that no development methodology is either the required or the implied standard. Following each checklist is a textual explanation of the items on the list.

1.3 Key Terms

Several key words appear throughout the text of this appendix. These words have specific meanings within the context of certified systems and their definitions are presented here. These definitions are taken directly from the TCSEC:

Access - A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

Audit Trail - A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions.

Covert Channel - A communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Data - Information with a specific physical representation.

Discretionary Access Control - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Mandatory Access Control - A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

Object - A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc.

Sensitivity Label - A piece of information that represents the security level of an object and that describes the sensitivity (e.g., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions.

Subject - An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

Trusted Computing Base (TCB) - The totality of protection mechanisms within a computer system -- including hardware firmware, and software -- the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Additional Term

This term is included because it appears frequently in the following text.

Security - The protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure. Security also pertains to personnel, data, communications, and the physical protection of computer installations [IEEE 1983]. Specifically, for the purposes of this report, security is defined by the criteria in the TCSEC, i.e., a given security problem corresponds with a specific TCSEC criteria.

1.4 Format of This Appendix

The format of this appendix follows the format of Part I of the TCSEC. Each Class of TCB is described through a direct quotation of the description from the TCSFC. After this description, each of the four subheadings, Security Policy, Accountability, Assurance, and Documentation, follows with its individual subheadings, such as Discretionary Access Control. For each of these, a synopsis of the certification criteria is presented, followed by a checklist of activities to be performed at each of the five phases of the software development life cycle. In those instances where one or more of the five phases is not presented, no special consideration needs to be made during this phase of the development. In addition, the checklists for the various subheadings are upwardly compatible, i.e., Requirements for Discretionary Access Control for a class C1 TCB also apply to the Requirements for Discretionary Access Control for a class C2 TCB. When these checklist items are initially introduced, they are presented in **bold-underline** and a textual explanation for each is given. When they are repeated, they are prefixed by "(X):," where X is the Class of TCB in which they were introduced checklist items preceded by an "o" come directly from the TCSEC; those preceded by a "-" were identified by this research.

Bold-Underline is used to indicate certification criteria and checklist items not contained in a lower class or changes and additions to already defined certification criteria or checklist items. Where there are no **bold-underline**, information has been carried over from lower classes without addition or modification. Also, the paragraph numbers from this document correspond exactly to those in the TCSEC to assist the user of this document in tracing a requirement back to its origin in the TCSEC.

2.0 DIVISION C: DISCRETIONARY PROTECTION

2.1 CLASS C1: DISCRETIONARY SECURITY PROTECTION

"The Trusted Computing Base (TCB) of a class C1 system nominally satisfies the discretionary security requirements by providing separation of users and data. It incorporates some form of credible controls capable of enforcing access limitations on an individual basis, i.e., ostensibly suitable for allowing users to be able to protect project or private information and to keep other users from accidentally reading or destroying their data. The class C1 environment is expected to be one of cooperating users processing data at the same level(s) of sensitivity."

2.1.1 Security Policy

2.1.1.1 Discretionary Access Control

- o <u>TCB shall define and control access between named users and named objects in the ADP system</u>
- o <u>Enforcement mechanism shall allow users to specify and control sharing of those objects by named individuals, or defined groups, or both</u>

Requirements:

- <u>Identify all types of users for the system, including individual users and groups of users</u>
- <u>Identify all types of objects (e.g., files and programs) in the system</u>
- <u>Identify the level of sensitivity for the data within the system</u>
- <u>Describe all possible interactions between the users and the data</u>
- <u>Identify the criteria for determining the need of a particular user to access a particular object</u>

To properly develop the Discretionary Access Control Requirements for a Class C1 TCB, the system developer must completely evaluate the system to be developed. This evaluation must identify all of the types of user objects that are to be handled by the system. In addition, all interactions between these users and objects need to be described. Once this information has been established, the security policy requirements for each type of interaction will be determined, and appropriate system requirements will be developed to reflect this determination.

Consider the case of the development of a secure database system. In particular, consider the user of type data entry person, and the data of type salary information. This data entry person may be allowed to enter the employee name and identification number; however, the annual salary associated with that employee would be sensitive information, and as such would not be made available to the data entry person. In this instance, the security policy requirements for this type of interaction would reflect this.

Design:

- Establish a method for maintaining the enforcement mechanism (e.g., self/group/public controls, access control lists)
- Establish a method of controlling access to objects within the domain of the TCB
- Establish a criteria for determining the need of a particular user to access a particular object
- Establish a mechanism for identifying users

The design of the discretionary access control for a Class C1 TCB must satisfy the requirements stated above. In particular, a method for enforcing the access control must be established. In addition, a mechanism for controlling access to objects within the domain of the TCB needs to be established, and the design needs to include a means of identifying users. For example, the discretionary access design may use passwords to control the access of users to objects in the TCB. Only those users on a given access list would be informed of the password required to gain access to data to which the list corresponds.

Coding:

- Use a defensive programming methodology, including hooks to aid testing and maintenance

The coding must implement the design of the discretionary access control of the Class C1 TCB. Coding should be performed defensively, using modular structured programming, as discussed in the introduction. The hooks typically are prudently placed debug statements that provide information on the operation of the system. A convenient means should be provided to turn the debug statements on and off.

Support:

- Re-test system upon completion of modification
- Ensure that enforcement mechanism access control lists are maintained, e.g., adding new users to the access control lists, and removing users from the access control lists who no longer require access to the system

Support for discretionary access control involves the maintenance of the system such as adding enhancements to it or removing obsolete features. The support includes the maintenance of the access control list by adding new users and removing users who no longer require access to the system.

2.1.2 Accountability

2.1.2.1 Identification and Authentication

- o <u>TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate</u>
- o <u>TCB shall use a protected mechanism to authenticate a user's identity</u>
- o <u>TCB shall protect authentication data so that it cannot be accessed by any unauthorized user</u>

Requirements:

- <u>Identify all actions to be mediated by the TCB, including access to authentication data</u>
- <u>Define a standard format for authentication data</u>
- <u>Determine a method for identifying users</u>

The major criterion to be met when establishing the requirements for Identification and Authentication of a Class C1 TCB is that users must identify themselves to the TCB before beginning to perform any actions to be mediated by the TCB. To accurately develop requirements for this, the developer needs to first establish a method for identifying users, and second establish a list of all actions to be mediated by the TCB.

The method for identifying users would vary depending upon the number of users requiring access or the sensitivity of the data within the system. For example, if the system contained very little private information, the developer may decide to implement somewhat trivial identification procedures for those actions that access the non-private information and more restrictive identification procedures for those actions that access the private information. If all of the information were private, restrictive identification procedures would be used in all cases. Regardless of the identification procedures used, authentication data would be used to verify the user's identify. The format for this authentication data (e.g., a password and a social security number) will need to be determined at this time.

Design:

- <u>Establish a protected mechanism for identification and authentication of users</u>
- <u>Establish a mechanism for creating and maintaining authentication data</u>

The Design of the Identification and Authentication aspect of a Class C1 TCB must be responsible for the actual identification of the users. To do this, a mechanism for identifying the users and authenticating their identification must be designed. Furthermore, a mechanism for creating, maintaining, and protecting the authentication data needs to be designed to guarantee that the data is properly maintained and is tamper-proof. This mechanism will be protected by the TCB, so that no user can accidentally access it and read, modify, or delete important authentication data.

2.1.3 Assurance

2.1.3.1 Operational Assurance

2.1.3.1.1 System Architecture

- o <u>TCB shall maintain a domain for its own execution that protects it from external interference or tampering</u>
- o <u>Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system</u>

Requirements:

- <u>Identify all resources to be protected by the TCB including the TCB code and data structures</u>

To develop the requirements for the System Architecture in a Class C1 TCB, all resources that the TCB must protect need to be identified. These resources include the code, data structures, and files in the TCB's domain. Protecting the resources will serve to isolate them from external interference and tampering, and thus ensure their integrity. Resources may require various methods of protection.

Design:

- <u>Establish a mode for protecting each resource within the TCB</u>

The System Architecture design for a Class C1 TCB must establish a mode for protecting each resource within the TCB. Such a mode may use security mechanisms, such as passwords, to restrict access to resources like its code. Hardware security mechanisms may be used to provide protection for some resources (e.g., descriptors that identify the security attributes of the subject and object, and gates that control access to resources).

2.1.3.1.2 System Integrity

- o <u>Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB</u>

Requirements:

- <u>Identify hardware and firmware elements of the TCB to be validated</u>
- <u>Determine validation criteria for testing each element of the TCB</u>
- <u>Determine the appropriate interval at which validation of the hardware/firmware should take place</u>

To correctly detail requirements for System Integrity of a Class C1 TCB, the system developer must identify all on-site hardware and firmware elements of the TCB. In addition, the criteria for determining validity of these elements need to be established. When establishing the validation criterion, each hardware and firmware element needs to be considered individually, rather than by type of element, because the validation of these elements is dependent upon their operation environment. As an example of this, consider a tape drive on the system. If this tape drive were used to backup a file containing non-sensitive data, its validation criteria would be less strict than those developed if it were used to back up sensitive data; however, the type of tape drive would be the same in both instances.

Design:

- <u>Establish a method for testing each criterion for each element of the TCB</u>

As described in the System Integrity Requirements, it is important to consider not only the type of the hardware being validated, but also its place in the operation of the TCB. Because of this, it is necessary, during the design phase, for the developer to completely understand each validation criteria for each piece of hardware and to design each test individually. While it is true that some tests may be able to be used for evaluating similar criteria on similar pieces of hardware, the criteria need to be evaluated separately and tests created separately to ensure that, once developed, they are complete and test exactly what needs to be validated.

2.1.3.2 Life-Cycle Assurance

2.1.3.2.1 Security Testing

- o <u>Security mechanisms - the ADP system shall be tested and found to work as claimed in the system documentation</u>
- o <u>Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB</u>

Requirements:

- <u>Identify security protection mechanisms of the TCB to be tested</u>

The performance of security testing in a Class C1 TCB ensures the integrity of the system's security. The system documentation serves as the basis for identifying the security protection mechanisms (e.g., discretionary access control) to be tested. To perform this testing, all security protection mechanisms with the TCB need to be identified.

Design:

- <u>Establish a means of testing each security protection mechanism</u>

The design of the security testing features for a Class C1 TCB must satisfy the requirement stated above. This is confirmed by demonstrating that the system's security complies with the system documentation. This design needs to provide a means of testing each security protection mechanism for which it is responsible. This may be accomplished by examining the data that results from the execution of diagnostic testing.

Coding:

- <u>Use prudently placed debug statements to allow the tester to monitor the operation of the security mechanisms</u>

The coding must implement the design of security testing features of the Class C1 TCB. In particular, the diagnostics may be derived from prudently placed debug statements to allow the tester to monitor the operation of the system's security.

2.1.4 Documentation

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for a TCB. The documentation must be developed to meet all requirements set forth in the TCSEC and must be complete and up-to-date; however, this is not particular to this development and requires no further discussion.

2.1.4.1 Security Features User's Guide

- o <u>Single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another</u>

2.1.4.2 Trusted Facility Manual

- o <u>Manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility</u>

2.1.4.3 Test Documentation

- o <u>System developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing</u>

2.1.4.4 Design Documentation

- o <u>Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB</u>
- - <u>If the TCB is composed of distinct modules, the interfaces between these modules shall be described</u>

2.2 CLASS C2: CONTROLLED ACCESS PROTECTION

"Systems in this class enforce a more finely grained discretionary access control than C1 systems, making users individually accountable for their actions through login procedures, auditing of security-relevant events, and resource isolation."

2.2.1 Security Policy

2.2.1.1 Discretionary Access Control

(C1):

- o TCB shall define and control access between named users and named objects in the ADP system
- o Enforcement mechanism shall allow users to specify and control sharing of those objects by named individuals or defined groups of individuals, or by both, and shall provide controls to limit propagation of access rights
- o Discretionary access control mechanism shall, either by explicit user action or default, provide that objects are protected from unauthorized access
- o Access controls shall be capable of including or excluding access to the granularity of a single user
- o Access permission to an object by users not already possessing access permission shall only be assigned by authorized users

Requirements:

(C1):

- Identify all types of users for the system, including individual users and groups of users
- Identify all types of objects (e.g., files and programs) in the system
- Identify the level of sensitivity for the data within the system
- Describe all possible interactions between the users and the data
- Identify the criteria for determining the need of a particular user to access a particular object
- Identify specific individuals to be included in each group of users

As can be seen in the description of a Class C2 TCB, this class requires a more finely grained discretionary access control than the Class C1. This degree of granularity is enforced by requiring that specific individuals must be included in a group and then identified during the requirements phase of the software development. In Class C2, the individual is accountable for his own actions even though he may be operating as a member of a group; therefore, it is still important to be able to identify the individual.

Design:

(C1):

- Establish a method for maintaining the enforcement mechanism (e.g., self/group/public controls, access control lists)
- Establish a method of controlling access to objects within the domain of the TCB
- Establish criteria for determining the need of a particular user to access a particular object
- Establish a mechanism for identifying users

Coding:

(C1):

- Use a defensive programming methodology, including hooks, to aid testing and maintenance

Support:

(C1):

- Re-test system upon completion of modification
- Ensure that enforcement mechanism access control lists are maintained, e.g., adding new users to the access control lists and removing users from the access control lists who no longer require access to the system

2.2.1.2 Object Reuse

o TCB shall assure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized

Requirements:

- Identify all situations in which a storage object is initially assigned, allocated, or reallocated from the TCB's pool of unused storage objects
- Identify methods for removing data from objects

Various types of storage objects controlled by a Class C2 TCB are subject to being reused; therefore, they need to be identified and the situations that would result in their reuse need to be identified. These situations include the initial assignment to, allocation to, and reallocation from the TCB's pool of unused storage objects.

In addition to identifying all situations in which an object could potentially be reused, the various methods for removing unauthorized data from those objects need to be identified. Once the possible methods for the removal of data have been identified, they can be evaluated against the other system requirements, such as those for system performance. In this manner, the best data removal method can be chosen and implemented during the design phase.

Design:

- <u>Establish a method for determining authorization of subject for object</u>
- <u>Establish a method for removing data for which subject is not authorized</u>

A mechanism for managing storage object reuse must be established. Its design must satisfy the requirements stated above, allow the determination of authorization of subjects for objects, and facilitate the removal of unauthorized data.

2.2.2 Accountability

2.2.2.1 Identification and Authentication

(C1):

- o TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate
- o TCB shall use a protected mechanism to authenticate the user's identity
- o TCB shall protect authentication data so that it cannot be accessed by any unauthorized user
- o <u>TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user</u>
- o <u>TCB shall provide the capability of associating the individual identity with all auditable actions taken by that individual</u>

Requirements:

(C1):

- Identify all actions to be mediated by the TCB, including access to authentication data
- Define a standard format for authentication data
- Determine a method for identifying users

Design:

(C1):

- Establish a mechanism for identifying and authenticating users
- Establish a mechanism for creating and maintaining authentication data
- Establish a mechanism for associating user identity with user actions

Under Class C1, the user must identify himself to the TCB prior to being allowed to access any actions mediated by it. This requirement is enforced in a C2 TCB, with the additional stipulation that all auditable actions performed by that user can be associated with that user. To accomplish this, some type of mechanism for associating the user's identity with the actions taken by that user must be available. To do this, during the design phase of Identification and Authentication, the mechanism will need to be detailed. As an example, the method chosen to identify the user could be to require him to enter his Social Security Number (SSN). The mechanism for associating the user identity with the user actions could create files containing SSN and actions for each auditable function mediated by the TCB. These files could be examined at a later time to check to see that all accesses to that function were performed in good faith and that no user accidentally read, modified, or deleted information which should not have been read, modified, or deleted by that user.

2.2.2.2 Audit

- o TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects
- o Audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data
- o TCB shall be able to record: use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events.

- o <u>For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record shall include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion events, the audit record shall include the name of the object</u>.
- o <u>ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity</u>

Requirements:

- <u>Identify all objects to be protected by the TCB</u>
- <u>Determine complete requirements for Audit Function</u>

To correctly determine the requirements for Class C2 TCB auditing functions, all objects that need to be protected by the TCB must be identified, and the means for auditing them must be determined. The use of identification and authentication data and the introduction of objects into a user's address space must be included. The following items must be monitored in order to audit each recorded event: the date and time of the event, user, type of event, and success or failure of the event.

Design:

- <u>Establish a mechanism for operation of the Audit Function</u>

The design of the auditing functions of a Class C2 TCB must satisfy the requirements stated above. To accomplish this, the design must include a mechanism for the operation of the audit functions. The audit record should be maintained on-line, and it should also be able to be output in a human-readable form.

2.2.3 Assurance

2.2.3.1 Operational Assurance

2.2.3.1.1 System Architecture

(C1):

- o TCB shall maintain a domain for its own execution that protects it from external interference or tampering
- o Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system

o <u>TCB shall isolate the resources to be protected so that they are subject to the access control and auditing requirements</u>

Requirements:

(C1):

- Identify all resources to be protected by the TCB, including the TCB code and data structures

Design:

(C1):

- Establish a mode for protecting each resource within the TCB
- <u>Establish a method of isolating resources to be protected by the TCB</u>

The design of the system architecture of a Class C2 TCB must satisfy the requirement stated above. In addition, the design must include a method for isolating resources to be protected by the TCB. The method may use a mechanism such as passwords for individuals or groups of individuals to further ensure the security of the resources by having their access controlled and audited. Thus, it provides additional protection of the resources from being accessed by unauthorized subjects.

2.2.3.1.2 System Integrity

(C1):

o Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB

Requirements:

(C1):

- Identify hardware and firmware elements of the TCB to be validated
- Determine validation criteria for testing each element of the TCB
- Determine the appropriate interval at which validation of the hardware/firmware should take place

Design:

(C1):

- Establish a method for testing each criterion for each element of the TCB

2.2.3.2 Life-Cycle Assurance

2.2.3.2.1 Security Testing

(C1):

- o Security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation
- o Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB
- o <u>Testing shall include a search for obvious flaws that would allow violation of resource isolation or that would permit unauthorized access to audit or authentication data</u>

Requirements:

(C1):

- Identify security protection mechanisms of the TCB to be tested

Design:

(C1):

- Establish a means of testing each security protection mechanism

Coding:

(C1):

- Use prudently placed debug statements to allow the tester to monitor operation of the security mechanisms

2.2.4 Documentation

(C1):

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB. because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for a TCB. The documentation must be developed to meet all requirements set forth in the TCSEC, and must be complete and up-to-date; however, this is not particular to this development and requires no further discussion.

2.2.4.1 Security Features User's Guide

(C1):

o Single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another

2.2.4.2 Trusted Facility Manual

(C1):

o Manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility.
o <u>The procedures for examining and maintaining the audit files, as well as the detailed audit record structure for each type of audit event, shall be given</u>.

2.2.4.3 Test Documentation

(C1):

o System developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing

2.2.4.4 Design Documentation

(C1):

- o Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB
- o If the TCB is composed of distinct modules, the interfaces between these modules shall be described

3.0 DIVISION B: MANDATORY PROTECTION

3.1 CLASS B1: LABELED SECURITY PROTECTION

"Class B1 systems require all the features required for Class C2. In addition, an informal statement of the security policy model, data labeling, and mandatory access control over named subjects and objects must be present. The capability must exist for accurately labeling exported information. Any flaws identified by testing must be removed."

3.1.1 Security Policy

3.1.1.1 Discretionary Access Control

(C1):

- o TCB shall define and control access between named users and named objects in the ADP system
- o Enforcement mechanism shall allow users to specify and control sharing of those objects by named individuals or defined groups of individuals, or both, and shall provide controls to limit propagation of access rights.

(C2):

- o Discretionary access control mechanism shall, either by explicit user action or default, provide that objects are protected from unauthorized access
- o Access controls shall be capable of including or excluding access to the granularity of a single user
- o Access permission to an object by users not already possessing access permission shall only be assigned by authorized users

Requirements:

(C1):

- Identify all types of users for the system, including individual users and groups of users
- Identify all types of objects (e.g., files and programs) in the system
- Identify the level of sensitivity for the data within the system
- Describe all possible interactions between the users and the data
- Identify the criteria for determining the need of a particular user to access a particular object

(C2):

- Identify specific individuals to be included in each group of users

Design:

(C1):

- Establish a method for maintaining the enforcement mechanism (e.g., self/group/public controls, access control lists)
- Establish a method of controlling access to objects within the domain of the TCB
- Establish criteria for determining the need of a particular user to access a particular object
- Establish a mechanism for identifying users

Coding:

(C1):

- Use a defensive programming methodology, including hooks, to aid testing and maintenance

Support:

(C1):

- Re-test system upon completion of modification
- Ensure that enforcement mechanism access control lists are maintained, e.g., adding new users to the access control lists and removing users from the access control lists who no longer require access to the system

3.1.1.2 Object Reuse

(C2):

o TCB shall ensure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized

Requirements:

(C2):

- Identify all situations in which a storage object is initially assigned, allocated, or reallocated from the TCB's pool of unused storage objects
- Identify methods for removing data from objects

Design:

(C2):

- Establish a method for determining authorization of subject for object
- Establish a method for removing data for which subject is not authorized

3.1.1.3 Labels

- o <u>Sensitivity labels associated with each subject and storage object under its control shall be maintained by the TCB</u>
- o <u>These labels shall be used as the basis for mandatory access control decisions</u>
- o <u>To import non-labeled data, the TCB shall request and receive from an authorized user the security level of the data, and all such actions shall be auditable by the TCB</u>

Requirements:

- <u>Identify all subjects and storage objects under the control of the TCB</u>
- <u>Define a policy for associating sensitivity labels with each subject and storage object controlled by the TCB, including the import of non-labeled data and the export of labeled data</u>

Sensitivity labels, as required for a Class B1 TCB, must be associated with each subject and storage object under control of the TCB. The labels shall be used as the basis for access control decisions; therefore, the subjects and storage objects requiring sensitivity labels must be identified, and a policy must be defined for associating the labels with each subject and storage object. This includes the importing of non-labeled data and the exporting of labeled data. Two typical types of data that might be labeled and protected by the TCB are personnel records and inventory data.

Design:

- Establish a mechanism for implementing and managing the sensitivity labels
- Establish an interaction between sensitivity labeling and the audit function
- Establish a mechanism for changing and monitoring the sensitivity designation

The design of the mechanism to handle sensitivity labels must satisfy the requirements stated above. To accomplish this it must have a means for implementing and managing the sensitivity labels. This involves handling the interactions between the sensitivity labeling and the audit functions. Also, the design requires a mechanism for changing and monitoring the sensitivity designation. The labels may be in the form of designators, such as distribution restrictors or enablers, for controlling access by individuals or groups of users and/or access type restrictors that limit the type of access permitted to an object. This provides the TCB with a means of maintaining the integrity of the security label information.

3.1.1.3.1 Label Integrity

- o Sensitivity labels shall accurately reflect security levels of the specific subjects or objects with which they are associated
- o Sensitivity labels shall accurately reflect the internal labels and shall be associated with exported information

Requirements:

- Identify the data required for sensitivity labels to represent, accurately and unambiguously, security levels of specific subjects or objects with which they are associated

Ensuring the integrity of the sensitivity labels requires the identification of the data needed to accurately and unambiguously represent security levels of specific subjects or objects with which they are associated. An analysis must be done to determine this data's critical characteristics that satisfy this criteria. Some typical characteristics are the security level associated with the subjects and objects, the group of users to be allowed access to objects, and restrictions on the mode of accessing an object (e.g., read access only, write access, privilege to purge, and monitor history of accesses).

Design:

- Establish an association between sensitivity labels exported by the TCB and the information being exported, such that the sensitivity labels accurately and unambiguously represent security levels of specific subjects or objects with which they are associated.

The design of the mechanism for maintaining the label integrity must satisfy the requirements stated above. Thus, it must include an accurate and unambiguous association between the sensitivity labels and the security levels of specific subjects or objects. This association must provide a secure and reliable logical connection between the label and its associated subject or object.

3.1.1.3.2 Exportation of Labeled Information

- o TCB shall designate each communication channel and I/O device as either single-level or multilevel; any change in this designation shall be done manually and shall be auditable by the TCB
- o TCB shall maintain and be able to audit any change in the current security level associated with a single communication channel or I/O device
- o When TCB exports an object to a multilevel I/O device, the sensitivity label associated with that object shall also be exported and shall reside on the same physical medium as the exported information and shall be in the same form
- o Export or import protocol used on a multilevel communication channel shall provide for the unambiguous pairing between the sensitivity labels and the associated information
- o Single-level I/O devices and single-level communication channels are not required to maintain the sensitivity labels of the information they process; however, the TCB shall include a mechanism by which the TCB and an authorized user reliably communicate to designate the single security level of imported or exported information
- o ADP system administrator shall be able to specify printable names associated with exported sensitivity labels
- o TCB shall mark the beginning and end of all human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the sensitivity of the output
- o TCB shall, by default, mark the top and bottom of each page of human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the overall sensitivity level of the page
- o TCB shall, by default, mark other forms of human-readable output with human-readable sensitivity labels that properly represent the sensitivity of the output
- o TCB shall audit any override of these marking defaults

Requirements:

- Identify the types of communication channels and I/O devices to be used with the TCB and whether they are single-level or multilevel devices
- Define the policy for establishing/changing the security designation associated with these devices
- Define the policy for handling the preparation and handling of human-readable output, including its form

A means must be provided for the exportation of labeled information. This requires the identification of the types of communication channels and I/O devices to be used with the TCB, including whether they are single-level or multilevel devices. Ensuring the proper management of this process requires the definition of the policy for establishing/changing the security designation associated with these devices. Also, a policy must be defined for the preparation and handling of human-readable output. These requirements promote secure and reliable transfer of labeled information through communication channels and between I/O devices.

Design:

- Establish the protocol for importing and exporting objects between the TCB and secure single-level or multilevel devices
- Establish the mechanism for labeling and producing human-readable output

The design of mechanisms for handling the exportation of labeled information must satisfy the requirements stated above. To accomplish this a protocol must be established for importing and exporting objects between the TCB and secure single-level or multilevel devices. Also for human-readable output (e.g., printed output and information displayed on a terminal), the design must include mechanisms for labeling and producing the output.

3.1.1.4 Mandatory Access Control

- o TCB shall enforce a mandatory access control policy over all subjects and storage objects under its control
- o All subjects and storage objects shall be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels shall be the basis of the mandatory access control decisions
- o TCB shall be able to support two or more security levels

Requirements:

- Identify all subjects and storage objects under the control of the TCB
- Describe the hierarchical classification levels and non-hierarchical categories within the TCB
- Define the mandatory access control policy based upon the classification levels within the TCB

To accurately define the requirements of the mandatory access control for a Class B1 TCB, all subjects and storage objects that are to be under the control of the TCB must first be identified. Then the hierarchical classification levels and non-hierarchical categories, which are to be associated with these subjects and objects within the TCB, need to be described. A policy for the mandatory access control must be defined that is based on these classification levels and categories. This policy will establish secure practices for subjects to access objects (e.g., read, write, and modify) for which they have sufficient clearance.

Design:

- Establish a mechanism for implementing the mandatory access control policy

The design of the mandatory access control must satisfy the requirements for a Class B1 TCB, as stated above. Thus, a mechanism must be established for implementing the mandatory access control policy. In particular, monitoring of subjects attempting to access objects needs to be performed such that the following hold for all accesses between subjects and objects controlled by the TCB: a subject may read an object only if its security level is great enough to access the object, as determined by comparison with the object's security level; subject may write or modify an object only if its security level is low enough to access the object, as determined by comparison with the object's security level. For further elaboration on mandatory access control, refer to the TCSEC.

3.1.2 Accountability

3.1.2.1 Identification and Authentication

(C1):

- o TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate
- o TCB shall maintain authentication data that includes information for verifying the identity of individual users as well as information for determining the clearance and authorizations of individual users

- o TCB shall use <u>this data to authenticate</u> the user's identity <u>and to determine the security level and authorizations of subjects that may be created to act on the behalf of the individual user</u>
- o TCB shall protect authentication data so that it cannot be accessed by any unauthorized user

(C2):

- TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user
- TCB shall provide the capability of associating the individual identity with all auditable actions taken by that individual

Requirements:

(C1):

- Identify all actions to be mediated by the TCB, including access to authentication data
- Define a standard format for authentication data
- Determine a method for identifying users

Design:

(C1):

- Establish a mechanism for identification and authentication of users
- Establish a mechanism for creating and maintaining authentication data

(C2):

- Establish a mechanism for associating user identity with user actions

3.1.2.2 Audit

(C2):

- o TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects
- o Audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data
- o TCB shall be able to record: use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system

administrators and/or system security officers, and other security relevant events.

- o **TCB shall be able to audit any override of human-readable output markings.**
- o For each recorded event, the audit record shall identify: date and time of event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record shall include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion events, the audit record shall include the name of the object **and the object's security level.**
- o ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity **and/or object security level.**

Requirements:

(C2):

- Identify all objects to be protected by the TCB
- Establish complete requirements for Audit Function

Design:

(C2):

- Establish a mechanism for operation of the Audit Function

3.1.3 Assurance

3.1.3.1 Operational Assurance

3.1.3.1.1 System Architecture

(C1):

- o TCB shall maintain a domain for its own execution that protects it from external interference or tampering
- o Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system
- o **TCB shall maintain process isolation through the provision of distinct address spaces under its control**

(C2):

o TCB shall isolate the resources to be protected so that they are subject to the access control and auditing requirements

Requirements:

(C1):

- Identify all resources <u>and distinct address spaces</u> to be protected by the TCB, including the TCB code and data structures

With one exception, the requirements for a Class B1 TCB will be developed in a manner that is essentially the same as that for the previous classes of TCBs. In a Class B1 TCB, the isolation of processes maintained by the TCB will be accomplished through the use of distinct address spaces to be controlled by the TCB. As a result of this exception, the requirements for this TCB will need to identify the address spaces to be used for this isolation. This identification is necessary so that the system developer can determine, during the design phase, the best way to protect the address space and its associated processes.

Design:

(C1):

- Establish a mode for protecting each resource <u>and each distinct address space</u> within the TCB

(C2):

- Establish a method of isolating resources to be protected by the TCB

As mentioned in the System Architecture Requirements, once the address spaces to be protected by the TCB have been identified, it is necessary for the system developer to establish a mode for protecting each of them. This protection scheme may include hardware, firmware, or software, or some combination of all three. Whatever the case, the intended use of the item to be protected needs to be determined so that the most reliable protection scheme is utilized.

3.1.3.1.2 System Integrity

(C1):

- o Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB

Requirements:

(C1):

- Identify hardware and firmware elements of the TCB to be validated
- Determine validation criteria for testing each element of the TCB
- Determine the appropriate interval at which validation of the hardware/firmware should take place

Design:

(C1):

- Establish a method for testing each criterion for each element of the TCB

3.1.3.2 Life-Cycle Assurance

3.1.3.2.1 Security Testing

(C1):

- o Security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation
- o Team of individuals who thoroughly understand the specific implementation of the TCB shall subject its design documentation, source code, and object code to thorough analysis and testing. The team's objectives will be to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy, and to assure that no subject is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users
- o All discovered flaws shall be removed or neutralized and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced

Requirements:

(C1):

- Identify security protection mechanisms of the TCB to be tested

Design:

(C1):

- Establish a means of testing each security protection mechanism <u>in a manner consistent with the stringent requirements detailed in the TCSEC</u>

Although the design of the Security Testing aspect of a Class B1 TCB is not really different from that of any of the previous classes, the requirements set forth for it in the TCSEC are much more detailed. As such, more attention needs to be paid to determining the means for testing each security protection mechanism. The design for this security testing needs to be performed in strict accordance with the requirements of the TCSEC.

Coding:

(C1):

- Use prudently placed debug statements to allow the tester to monitor operation of the security mechanisms

3.1.3.2.2 Design Specification and Verification

o <u>Informal or formal model of the security policy supported by the TCB shall be maintained that is shown to be consistent with its axioms</u>.

Design:

- <u>Identify the model to be used</u>
- <u>Establish consistency between design and the model</u>

Although no actual software development needs to be performed to satisfy this aspect of a Class B1 TCB, some consideration needs to be made for it during the design phase for the system as a whole. During the Design Phase the system developer needs to identify a model of the security policy that can be used to establish consistency between the security policy and the system design. This consistency should be demonstrated prior to the commencement of the coding phase so that design modifications are eliminated in the latter stages of the system development.

3.1.4 Documentation

(C1):

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for a TCB. The documentation must be developed to meet all requirements set forth in the TCSEC, and must be complete and up-to-date; however, this is not particular to this development and requires no further discussion.

3.1.4.1 Security Features User's Guide

(C1):

o Single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another

3.1.4.2 Trusted Facility Manual

(C1):

o Manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility

(C2):

o The procedures for examining and maintaining the audit files, as well as the detailed audit record structure for each type of audit event, shall be given.

o Manual shall describe the operator and administrator functions related to security, to include changing the security characteristics of a user

o Manual shall provide guidelines on the consistent and effective use of the protection features of the system, how they interact, how to securely generate a new TCB, and facility procedures, warnings, and privileges that need to be controlled in order to operate the facility in a secure manner

3.1.4.3 Test Documentation

(C1):

- o System developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing

3.1.4.4 Design Documentation

(C1):

- o Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB
- o If the TCB is composed of distinct modules, the interfaces between these modules shall be described
- o <u>An informal or formal description of the security policy model enforced by the TCB shall be available and an explanation provided to show that it is sufficient to enforce the security policy</u>
- o <u>Specific TCB protection mechanisms shall be identified and an explanation given to show that they satisfy the model</u>

3.2 CLASS B2: STRUCTURED PROTECTION

"In class B2 systems, the TCB is based on a clearly defined and documented formal security policy model that requires the discretionary and mandatory access control enforcement found in class B1 systems be extended to all subjects and objects in the ADP system. In addition, covert channels are addressed. The TCB must be carefully structured into protection-critical and non-protection-critical elements. The TCB interface is well-defined and the TCB design and implementation enable it to be subjected to more thorough testing and more complete review. Authentication mechanisms are strengthened, trusted facility management is provided in the form of support for system administrator and operator functions, and stringent configuration management controls are imposed. The system is relatively resistant to penetration."

3.2.1 Security Policy

3.2.1.1 Discretionary Access Control

(C1):

- o TCB shall define and control access between named users and named objects in the ADP system
- o Enforcement mechanism shall allow users to specify and control sharing of those objects by named individuals or defined groups of individuals, or both, and shall provide controls to limit propagation of access rights.

(C2):

- o Discretionary access control mechanism shall, either by explicit user action or default, provide that objects are protected from unauthorized access
- o Access controls shall be capable of including or excluding access to the granularity of a single user
- o Access permission to an object by users not already possessing access permission shall only be assigned by authorized users

Requirements:

(C1):

- Identify all types of users for the system, including individual users and groups of users
- Identify all types of objects (e.g., files and programs) in the system
- Identify the level of sensitivity for the data within the system
- Describe all possible interactions between the users and the data

- Identify the criteria for determining the need of a particular user to access a particular object

(C2):

- Identify specific individuals to be included in each group of users

Design:

(C1):

- Establish the method for maintaining the enforcement mechanism (e.g., self/group/public controls, access control lists)
- Establish the method of controlling access to objects within the domain of the TCB
- Establish the criteria for determining the need of a particular user to access a particular object
- Establish mechanism for identifying users

Coding:

(C1):

- Use a defensive programming methodology, including hooks to aid testing and maintenance

Support:

(C1):

- Re-test system upon completion of modification
- Ensure that enforcement mechanism access control lists are maintained, e.g., adding new users to the access control lists and removing users from the access control lists who no longer require access to the system

3.2.1.2 Object Reuse

(C2):

o TCB shall assure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized

Requirements:

(C2):

- Identify all situations in which a storage object is initially assigned, allocated, or reallocated from the TCB's pool of unused storage objects
- Identify methods for removing data from objects

Design:

(C2):

- Establish a method for determining authorization of subject for object
- Establish a method for removing data for which subject is not authorized

3.2.1.3 Labels

o Sensitivity labels associated with each ADP system resource (e.g., subject, storage object) that is directly or indirectly accessible by subjects external to the TCB shall be maintained by the TCB.

(B1):

o These labels shall be used as the basis for mandatory access control decisions.
o To import non-labeled data, the TCB shall request and receive from an authorized user the security level of the data, and all such actions shall be auditable by the TCB

Requirements:

(B1):

- Identify all ADP system subjects and storage objects that are directly or indirectly accessible by subjects external to the TCB
- Define a policy for associating sensitivity labels with each ADP system subject and storage object that is directly or indirectly accessible by subjects external to the TCB, including the import of non-labeled data and the export of labeled data

Design:

(B1):

- Establish a mechanism for implementing and managing the sensitivity labels
- Establish an interaction between sensitivity labeling and the audit function
- Establish a mechanism for changing and monitoring the sensitivity designation

3.2.1.3.1 Label Integrity

(B1):

o Sensitivity labels shall accurately reflect security levels of the specific subjects or objects with which they are associated
o Sensitivity labels shall accurately reflect the internal labels and shall be associated with exported information

Requirements:

(B1):

- Identify the data required for sensitivity labels to accurately and unambiguously represent security levels of specific subjects or objects with which they are associated.

Design:

(B1):

- Establish an association between sensitivity labels exported by the TCB, with the information being exported, such that they accurately and unambiguously represent security levels of specific subjects or objects with which they are associated.

3.2.1.3.2 Exportation of Labeled Information

(B1):

o TCB shall designate each communication channel and I/O device as either single-level or multilevel; any change in this designation shall be done manually and shall be auditable by the TCB

- TCB shall maintain and be able to audit any change in the current security level associated with a single communication channel or I/O device
- When TCB exports an object to a multilevel I/O device, the sensitivity label associated with that object shall also be exported and shall reside on the same physical medium as the exported information and shall be in the same form
- Export or import protocol used on a multilevel communication channel shall provide for the unambiguous pairing between the sensitivity labels and the associated information
- Single-level I/O devices and single-level communication channels are not required to maintain the sensitivity labels of the information they process; however, the TCB shall include a mechanism by which the TCB and an authorized user reliably communicate to designate the single security level of imported or exported information
- ADP system administrator shall be able to specify printable names associated with exported sensitivity labels
- TCB shall mark the beginning and end of all human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the sensitivity of the output
- TCB shall, by default, mark the top and bottom of each page of human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the overall sensitivity level of the page
- TCB shall, by default, mark other forms of human-readable output with human-readable sensitivity labels that properly represent the sensitivity of the output
- TCB shall audit any override of these marking defaults

Requirements:

(B1):

- Identify the types of communication channels and I/O devices to be used with the TCB and whether they are single-level or multilevel devices
- Define the policy for establishing/changing the security designation associated with these devices
- Define the policy for handling the preparation and handling of human-readable output, including its form

Design:

(B1):

- Establish the protocol for importing and exporting objects between the TCB and secure single-level or multilevel devices
- Establish the mechanism for labeling and producing human-readable output

3.2.1.3.3 Subject Sensitivity Labels

- o <u>TCB shall immediately notify a terminal user of each change in the security level associated with that user during an interactive session</u>
- o <u>A terminal user shall be able to query the TCB as desired for a display of the subject's complete sensitivity label</u>

Requirements:

- <u>Identify a means for the TCB to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session</u>
- <u>Identify a means for a terminal user to query the TCB as desired for a display of the subject's complete sensitivity label</u>

To properly determine the requirements for subject sensitivity labels used in a Class B2 TCB, a means must be identified to convey information (about the subject sensitivity) between the user and the TCB. Thus, the TCB must be able to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session. In addition, a terminal user must be able to query the TCB as desired for a display of the subject's complete sensitivity label. This allows a subject to remain fully informed of the contents of his sensitivity label during a session, including the current security level associated with it.

Design:

- <u>Establish a mechanism for the TCB to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session</u>
- <u>Establish a mechanism for a terminal user to query the TCB as desired for a display of the subject's complete sensitivity label</u>

The design of the mechanism that handles subject sensitivity labels in a Class B2 TCB must satisfy the requirements stated above. Thus a mechanism must be established for the TCB to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session. In addition, a mechanism must be established for a terminal user to query the TCB, as desired, for a display of the subject's complete sensitivity label. These mechanisms provide the means of keeping the terminal user informed of the contents of his current sensitivity level.

3.2.1.3.4 Device Labels

- o TCB shall support the assignment of minimum and maximum security levels to all attached physical devices
- o Security levels shall be used by the TCB to enforce constraints imposed by the physical environments in which the devices are located

Requirements:

- Identify a means for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices

To properly determine the requirements for device labels in a Class B2 TCB, a means must be identified for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices. This promotes the maintenance of sufficiently secure usage of the physical devices controlled by the TCB.

Design:

- Establish a mechanism for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices

The design of the mechanism for handling device labels in a Class B2 TCB must satisfy the requirements stated above. To accomplish this a mechanism must be established for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices. This must ensure that the TCB controls its physical devices in a secure manner such that the security levels used by the TCB enforce constraints imposed by the physical environments in which the devices are located.

3.2.1.4 Mandatory Access Control

- o TCB shall enforce a mandatory access control policy over all resources (i.e., subjects, storage objects and I/O devices) that are directly or indirectly accessible by subjects external to the TCB

(B1):

- o All subjects and storage objects shall be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels shall be the basis of the mandatory access control decisions
- o TCB shall be able to support two or more security levels

Requirements:

(B1):

- Identify all <u>resources that are directly or indirectly accessible by subjects external to the TCB</u>
- Describe the hierarchical classification levels and non-hierarchical categories within the TCB
- Define the mandatory access control policy based upon the classification levels within the TCB

In developing the requirements for the Mandatory Access Control aspect of a Class B2 TCB, the control policy remains as for a Class B1 TCB; however, in a Class B2 TCB, the policy is applicable to all resources that are directly or indirectly accessible by subjects external to the TCB. As a result of this expansion of applicability, the developer must, during the requirements phase, identify all those resources that meet the criteria. Once the resources to be controlled have been identified, the system development will progress as before except that more resources need to be considered.

Design:

(B1):

- Establish a mechanism for implementing the mandatory access control policy

3.2.2 Accountability

3.2.2.1 Identification and Authentication

(C1):

o TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate

o TCB shall maintain authentication data that includes information for verifying the identity of individual users as well as information for determining the clearance and authorizations of individual users

o TCB shall use this data to authenticate the user's identity and to determine the security level and authorizations of subjects that may be created to act on the behalf of the individual user

o TCB shall protect authentication data so that it cannot be accessed by any unauthorized user

(C2):

- o TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user
- o TCB shall provide the capability of associating the individual identity with all auditable actions taken by that individual

Requirements:

(C1):

- Identify all actions to be mediated by the TCB, including access to authentication data
- Define a standard format for authentication data
- Determine a method for identifying users

Design:

(C1):

- Establish a mechanism for identification and authentication of users
- Establish a mechanism for creating and maintaining authentication data

(C2):

- Establish a mechanism for associating user identity with user actions

3.2.2.1.1 Trusted Path

- o <u>TCB shall support a trusted communication path between itself and user for initial login and authentication</u>
- o <u>Communication via this path shall be initiated exclusively by a user</u>

Requirements:

- <u>Identify a communication path between the user and the TCB that can be trusted</u>
- <u>Identify all operations required for the TCB to support the trusted communication path</u>

In identifying the requirements for the development of a trusted path for a Class B1 TCB, a number of preliminary determinations need to be made. First, the system developer needs to identify the particular trusted path to be used on the system under development. This is required prior to the next determination, the identification of all support operations for the trusted path, because different paths will be supported in different manners. For example, if the trusted path identified was through a terminal that was enclosed in a vault with an around-the-clock guard, the support function may not need to be as security conscious as if the trusted path was made through the use of a terminal in the middle of an unsecured room. For the terminal in the middle of the unsecured room, some type of front-end may need to be implemented to protect the path from tampering.

Design:

- <u>Establish a mechanism for supporting the trusted path</u>
- <u>Establish a mechanism for enabling a user to access the TCB through a trusted communication path which interfaces with the identification and authentication function</u>

Once the items in the Requirements phase have been identified, mechanisms to meet the requirements need to be designed. In particular, each support operation will need to be designed so that it will operate in a secure manner. In addition, a mechanism for allowing a user to utilize the trusted path to access the TCB will need to be established. This mechanism will need to interface with the identification and authentication function developed for this Class of TCB so that user security is guaranteed. Each of these mechanisms will need to be especially reliable so that the security of the TCB is not compromised.

3.2.2.2 Audit

(C2):

- o TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects
- o Audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data
- o TCB shall be able to record: use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events.
- o TCB shall be able to audit any override of human-readable output markings.
- o For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record shall include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion

events, the audit record shall include the name of the object and the object's security level.

- o ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity and/or object security level.
- o <u>TCB shall be able to audit the identified events that may be used in the exploitation of covert storage channels</u>

Requirements:

(C2):

- Identify all objects to be protected by the TCB
- Establish complete requirements for Audit Function

Design:

(C2):

- Establish a mechanism for operation of the Audit Function

3.2.3 Assurance

3.2.3.1 Operational Assurance

3.2.3.1.1 System Architecture

(C1):

- o TCB shall maintain a domain for its own execution that protects it from external interference or tampering

(B1):

- o TCB shall maintain process isolation through the provision of distinct address spaces under its control
- o <u>TCB shall be internally structured into well-defined largely independent modules</u>
- o <u>TCB shall make effective use of available hardware to separate those elements that are protection-critical from those that are not</u>
- o <u>TCB modules shall be designed such that the principle of least privilege is enforced</u>
- o <u>Features in hardware, such as segmentation, shall be used to support</u>

logically distinct storage objects with separate attributes (namely: readable, writable)

o TCB user-interface shall be completely defined and all elements of the TCB identified

Requirements:

(B1):

- Identify all resources and distinct address spaces to be protected by the TCB, including the TCB code and data structures
- Identify all ways in which a user can interface with the TCB by identifying all elements of the TCB and identifying the user interface to each element

To satisfy the criteria for the System Architecture of a Class B2 TCB, one additional requirement needs to be satisfied. To satisfy the requirement, the user interface to the TCB needs to be completely defined. First, all elements of the TCB need to be identified. Second, the user interface to each element needs to be identified. The complete definition of the user interface of the TCB will then be defined as the sum of all of these individual interactions. As an example, one of the elements of the TCB could be a file of data. The user interface to that file could only be accomplished through use of a program. Therefore one of the aspects of the user interface for the TCB would be the module in the program that enables the user access to that file.

In addition, requirements as to how the TCB is to be developed are made in the System Architecture requirements for a Class B2 TCB. Although these requirements add great detail to the System Architecture, they are simply statements of good Software Engineering practices which should be used in all systems development. As such, no additional items need to be identified.

Design:

(C2):

- Establish a method of isolating resources to be protected by the TCB

(B1):

- Establish a mode for protecting each resource and distinct address space within the TCB
- Establish the user interface for the TCB

Once the requirements for the System Architecture aspect of a Class B2 TCB have been identified, the design phase can commence. During the design phase, as discussed in Class B1, the system developer needs to establish a mode for protecting each resource and distinct address space within the TCB. In addition, the developer needs to design the user interface in accordance with the requirements set forth for the System Architecture. This design must not only facilitate those operations that should be allowed, but it must also prohibit those operations that should be disallowed.

3.2.3.1.2 System Integrity

(C1):

o Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB

Requirements:

(C1):

- Identify hardware and firmware elements of the TCB to be validated
- Determine validation criteria for testing each element of the TCB
- Determine the appropriate interval at which validation of the hardware/firmware should take place

Design:

(C1):

- Establish a method for testing each criterion for each element of the TCB

3.2.3.1.3 Covert Channel Analysis

o <u>System developer shall conduct a thorough search for covert storage channels and make a determination of the maximum bandwidth of each identified channel</u>

The Covert Channel Analysis for a Class B2 TCB involves a thorough search, by the system developer, for covert storage channels. This search is a purely manual process, although the results from the search and actions taken in response to the search need to be detailed in the system documentation, no software development needs to take place. As such, a discussion of the Requirements, Design, Coding, Testing, or Support phase is not necessary.

3.2.3.1.4 Trusted Facility Management

o <u>TCB shall support separate operator and administrator functions</u>

Requirements:

- <u>Identify all functions to be performed by the operator and administrator</u>

The Trusted Facility Management of a Class B2 TCB provides the system with requirements for separate operator and administrator functions. In order to properly develop requirements for this, the system developer needs to completely define the roles of operator and administrator. Because these roles are to be maintained as separate functions, they must be considered separately, and the actions to be performed by each must be identified. In addition, any interaction between the two functions needs to be described in detail.

Design:

- <u>Establish a mechanism for the proper operation of each of the functions for the operator and each of the functions for the administrator</u>

During the design phase of the development of the operator and the administrator functions, each of the actions to be performed by each of the roles needs to be considered as an individual entity. In this phase, the determination of how each of the actions will be performed will be made. During this phase, it is particularly important to ensure that the actions performed by the operator and administrator oversee the actions of all other users and can contribute to the maintenance of the trust and security of the system as a whole.

3.2.3.2 Life-Cycle Assurance

3.2.3.2.1 Security Testing

(C1):

o Security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation

(B1):

- o Team of individuals who thoroughly understand the specific implementation of the TCB shall subject its design documentation, source code, and object code to thorough analysis and testing. The team's objectives will be to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy, and to ensure that no subject is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users
- o TCB shall be found relatively resistant to penetration.
- o All discovered flaws shall be corrected and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced
- o Testing shall demonstrate that the TCB implementation is consistent with the descriptive top-level specification

Requirements:

(C1):

- Identify security protection mechanisms of the TCB to be tested

Design:

(B1):

- Establish means of testing each security protection mechanism in a manner consistent with the stringent requirements detailed in the TCSEC

Coding:

(C1):

- Use prudently placed debug statements to allow the tester to monitor operation of the security mechanisms

3.2.3.2.2 Design Specification and Verification

- o Formal model of the security policy supported by the TCB shall be maintained that is proven to be consistent with its axioms
- o A descriptive top-level specification (DTLS) of the TCB shall be maintained that completely and accurately describes the TCB in terms of exceptions, error messages, and effects. It shall be shown to be an accurate description of the TCB interface.

Design:

(B1):

- Identify the model to be utilized
- Prove consistency between design and the model

As discussed previously in Class B1, the Design Specification and Verification of a TCB does not require any software development; however, it does need to be considered during the design phase of the software development for the TCB as a whole. During the design phase, the model to be used must be chosen. In a Class B2 TCB, this is required to be a formal model. The consistency between the design and the chosen model needs to be formally proven. This formal proof entails the development of "a complete and convincing mathematical argument to present full logical justification for each proof step, for the truth of a theorem, or set of theorems." To satisfy this aspect of the development of a Class B2 TCB, this proof will need to be formalized during the design phase of the system development.

3.2.3.2.3 Configuration Management

- o A configuration management system shall be in place that maintains control of changes to the descriptive top-level specification (DTLS), other design data, implementation documentation, source code, the running version of the object code, and test fixtures and documentation
- o The configuration management system shall assure a consistent mapping among all documentation and code associated with the current version of the TCB
- o Tools shall be provided for generating a new version of the TCB from source code and for comparing a newly generated version with the previous TCB version in order to ascertain that only the intended changes have been made in the code that will actually be used as the new version of the TCB

The development of any large system requires the introduction and proper use of some type of Configuration Management System. The development of a TCB is no exception to this rule; however, as it pertains to our discussion, Configuration Management does not require any special considerations. A Configuration Management System, which will guarantee that the requirements for Configuration Management are satisfied, must be used.

3.2.4 Documentation

(C1):

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for a TCB. The documentation must be developed to meet all requirements set forth in the TCSEC, and must be complete and up-to-date; however, this is not particular to this development and requires no further discussion.

3.2.4.1 Security Features User's Guide

(C1):

o Single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another

3.2.4.2 Trusted Facility Manual

(C1):

o Manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility

(C2):

o The procedures for examining and maintaining the audit files, as well as the detailed audit record structure for each type of audit event, shall be given.

(B1):

o Manual shall describe the operator and administrator functions related to security, to include changing the security characteristics of a user
o Manual shall provide guidelines on the consistent and effective use of the protection features of the system, how they interact, how to securely generate a new TCB, and facility procedures, warnings, and privileges that need to be controlled in order to operate the facility in a secure manner

- <u>TCB modules that contain the reference validation mechanism shall be identified</u>
- <u>Procedures for secure generation of a new TCB from source after modification of any modules in the TCB shall be described</u>

3.2.4.3 Test Documentation

(C1):

- System developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing
- <u>Documentation shall include results of testing the effectiveness of the methods used to reduce covert channel bandwidths</u>

3.2.4.4 Design Documentation

(C1):

- Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB
- <u>The interfaces between the TCB modules shall be described</u>
- A <u>formal</u> description of the security policy model enforced by the TCB shall be available and <u>proven</u> that is sufficient to enforce the security policy
- Specific TCB protection mechanisms shall be identified and an explanation given to show that they satisfy the model
- <u>The DTLS shall be shown to be an accurate description of the TCB interface</u>
- <u>Documentation shall describe how the TCB implements the reference monitor concept and give an explanation why it is tamper resistant, cannot be bypassed, and is correctly implemented</u>.
- <u>Documentation shall describe how the TCB is structured to facilitate testing and to enforce least privilege</u>
- <u>Documentation shall also present the results of covert channel analysis and the tradeoffs involved in restricting the channels</u>
- <u>All auditable events that may be used in the exploitation of known covert storage channels shall be identified</u>
- <u>The bandwidths of known covert storage channels, the use of which is not detectable by the auditing mechanisms, shall be provided</u>

3.3 CLASS B3: SECURITY DOMAINS

"The class B3 TCB must satisfy the reference monitor requirements that it mediate all accesses of subjects to objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. A security administrator is supported, audit mechanisms are expanded to signal security-relevant events, and system recovery procedures are required. The system is highly resistant to penetration."

3.3.1 Security Policy

3.3.1.1 Discretionary Access Control

(C1):

- o TCB shall define and control access between named users and named objects in the ADP system
- o Enforcement mechanism <u>(e.g., access control lists)</u> shall allow users to specify and control sharing of those <u>objects</u>, and shall provide controls to limit propagation of access rights

(C2):

- o Discretionary access control mechanism shall, either by explicit user action or default, provide that objects are protected from unauthorized access
- o Access controls shall be capable of <u>specifying, for each named object, a list of named individuals and a list of groups of named individuals with their respective modes of access to that object; furthermore, for each such named object, it shall be possible to specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given</u>
- o Access permission to an object by users not already possessing access permission shall only be assigned by authorized users

Requirements:

(C1):

- Identify all types of users for the system, including individual users and groups of users
- Identify all types of objects (e.g., files and programs) in the system
- Identify the level of sensitivity for the data within the system
- Describe all possible interactions between the users and the data

- Identify the criteria for determining the need of a particular user to access a particular object

(C2):

- Identify specific individuals to be included in each group of users

Design:

(C1):

- Establish a method for maintaining the enforcement mechanism (e.g., self/group/public controls, access control lists)
- Establish a method of controlling access to objects within the domain of the TCB
- Establish criteria for determining the need of a particular user to access a particular object
- Establish a mechanism for identifying users

Coding:

(C1):

- Use a defensive programming methodology, including hooks, to aid testing and maintenance

Support:

(C1):

- Re-test system upon completion of modification
- Ensure that enforcement mechanism access control lists are maintained, e.g., adding new users to the access control lists and removing users from the access control lists who no longer require access to the system

3.3.1.2 Object Reuse

(C2):

o TCB shall assure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized

Requirements:

(C2):

- Identify all situations in which a storage object is initially assigned, allocated, or reallocated from the TCB's pool of unused storage objects
- Identify methods for removing data from objects

Design:

(C2):

- Establish a method for determining authorization of subject for object
- Establish a method for removing data for which subject is not authorized

3.3.1.3 Labels

(B2):

o Sensitivity labels associated with each ADP system resource (e.g., subject, storage object) that is directly or indirectly accessible by subjects external to the TCB shall be maintained by the TCB

(B1):

o These labels shall be used as the basis for mandatory access control decisions.
o To import non-labeled data, the TCB shall request and receive from an authorized user the security level of the data, and all such actions shall be auditable by the TCB

Requirements:

(B2):

- Identify all ADP system subjects and storage objects that are directly or indirectly accessible by subjects external to the TCB
- Define a policy for associating sensitivity labels with each ADP system subject and storage object that is directly or indirectly accessible by subjects external to the TCB, including the import of non-labeled data and the export of labeled data

Design:

(B1):

- Establish a mechanism for implementing and managing the sensitivity labels
- Establish an interaction between sensitivity labeling and the audit function
- Establish a mechanism for changing and monitoring the sensitivity designation

3.3.1.3.1 Label Integrity

(B1):

o Sensitivity labels shall accurately reflect security levels of the specific subjects or objects with which they are associated

o Sensitivity labels shall accurately reflect the internal labels and shall be associated with exported information

Requirements:

(B1):

- Identify the data required for sensitivity labels to accurately and unambiguously represent security levels of specific subjects or objects with which they are associated

Design:

(B1):

- Establish an association between sensitivity labels exported by the TCB, with the information being exported, such that they accurately and unambiguously represent security levels of specific subjects or objects with which they are associated

3.3.1.3.2 Exportation of Labeled Information

(B1):

o TCB shall designate each communication channel and I/O device as either single-level or multilevel; any change in this designation shall be done manually and shall be auditable by the TCB

- o TCB shall maintain and be able to audit any change in the current security level associated with a single communication channel or I/O device
- o When TCB exports an object to a multilevel I/O device, the sensitivity label associated with that object shall also be exported and shall reside on the same physical medium as the exported information and shall be in the same form
- o Export or import protocol used on a multilevel communication channel shall provide for the unambiguous pairing between the sensitivity labels and the associated information
- o Single-level I/O devices and single-level communication channels are not required to maintain the sensitivity labels of the information they process; however, the TCB shall include a mechanism by which the TCB and an authorized user reliably communicate to designate the single security level of imported or exported information
- o ADP system administrator shall be able to specify printable names associated with exported sensitivity labels
- o TCB shall mark the beginning and end of all human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the sensitivity of the output
- o TCB shall, by default, mark the top and bottom of each page of human-readable, paged, hardcopy output with human-readable sensitivity labels that properly represent the overall sensitivity level of the page
- o TCB shall, by default, mark other forms of human-readable output with human-readable sensitivity labels that properly represent the sensitivity of the output
- o TCB shall audit any override of these marking defaults

Requirements:

(B1):

- Identify the types of communication channels and I/O devices to be used with the TCB and whether they are single-level or multilevel devices
- Define the policy for establishing/changing the security designation associated with these devices
- Define the policy for preparing and handling human-readable output, including its form

Design:

(B1):

- Establish the protocol for importing and exporting objects between the TCB and secure single-level or multilevel devices
- Establish the mechanism for labeling and producing human-readable output

3.3.1.3.3 Subject Sensitivity Labels

(B2):

- o TCB shall immediately notify a terminal user of each change in the security level associated with that user during an interactive session
- o A terminal user shall be able to query the TCB as desired for a display of the subject's complete sensitivity label

Requirements:

(B2):

- Identify a means for the TCB to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session
- Identify a means for a terminal user to query the TCB as desired for a display of the subject's complete sensitivity label

Design:

(B2):

- Establish a mechanism for the TCB to automatically notify a terminal user of each change in the security level associated with a given user during an interactive session
- Establish a mechanism for a terminal user to query the TCB as desired for a display of the subject's complete sensitivity label

3.3.1.3.4 Device Labels

(B2):

- o TCB shall support the assignment of minimum and maximum security levels to all attached physical devices
- o Security levels shall be used by the TCB to enforce constraints imposed by the physical environments in which the devices are located

Requirements:

(B2):

- Identify a means for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices

Design:

(B2):

- Establish a mechanism for the TCB to support the assignment of minimum and maximum security levels to all attached physical devices

3.3.1.4 Mandatory Access Control

(B2):

o TCB shall enforce a mandatory access control policy over all resources (i.e., subjects, storage objects, and I/O devices) that are directly or indirectly accessible by subjects external to the TCB

(B1):

o All subjects and storage objects shall be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels shall be the basis of the mandatory access control decisions
o TCB shall be able to support two or more security levels

Requirements:

(B1):

- Describe the hierarchical and non-hierarchical classification levels within the TCB
- Define the mandatory access control policy based upon the classification levels within the TCB

(B2):

- Identify all resources that are directly or indirectly accessible by subjects external to the TCB

Design:

(B1):

- Establish a mechanism for implementing the mandatory access control policy

3.3.2 Accountability

3.3.2.1 Identification and Authentication

(C1):

- o TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate
- o TCB shall maintain authentication data that includes information for verifying the identity of individual users as well as information for determining the clearance and authorizations of individual users
- o TCB shall use this data to authenticate the user's identity and to determine the security level and authorizations of subjects that may be created to act on the behalf of the individual user
- o TCB shall protect authentication data so that it cannot be accessed by any unauthorized user

(C2):

- o TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user
- o TCB shall provide the capability of associating the individual identity with all auditable actions taken by that individual

Requirements:

(C1):

- Identify all actions to be mediated by the TCB, including access to authentication data
- Define a standard format for authentication data
- Determine a method for identifying users

Design:

(C1):

- Establish a mechanism for identifying and authenticating users
- Establish a mechanism for creating and maintaining authentication data

(C2):

- Establish a mechanism for associating user identity with user actions

3.3.2.1.1 Trusted Path

- o TCB shall support a trusted communication path between itself and users for use when a positive TCB-to-user connection is required (e.g., login, change subject security level)
- o Communication via this trusted path shall be activated exclusively by a user or the TCB and shall be logically isolated and unmistakably distinguishable from other paths

Requirements:

(B2):

- Identify a trusted communication path between the user and the TCB
- Identify all operations required for the TCB to support the trusted communication path

Design:

(B2):

- Establish a mechanism for supporting the trusted path
- Establish a mechanism for enabling user-to-TCB and TCB-to-user access via the trusted communication path which interfaces with the identification and authentication function

The Trusted Path for a Class B3 TCB needs to allow bi-directional access, from user to TCB and from TCB to user. This consideration must come into play when developing the design for the mechanism that will enable this access. All other items from the Class B2 TCB also apply here.

3.3.2.2 Audit

(C2):

- o TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects
- o Audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data
- o TCB shall be able to record: use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events.

- TCB shall be able to audit any override of human-readable output markings.
- o For each recorded event, the audit record shall identify: date and time of event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record shall include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion events, the audit record shall include the name of the object and the object's security level.
- o ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity and/or object security level.

(B2):

- o TCB shall be able to audit the identified events that may be used in the exploitation of covert storage channels
- o <u>TCB shall contain a mechanism that is able to monitor the occurrence or accumulation of security auditable events that may indicate an imminent violation of security policy; this mechanism shall be able to immediately notify the security administrator when thresholds are exceeded and, if the occurrence or accumulation of these security relevant events continues, the system shall take the least disruptive action to terminate the event</u>

Requirements:

(C2):

- Identify all objects to be protected by the TCB
- Establish complete requirements for Audit Function
- <u>Define criteria for indication of an imminent violation of security policy</u>

The requirements for the auditing functions in a Class B3 TCB must include those of Classes C2, B1, and B2. In addition, a criteria must be defined for the indication of an imminent violation of the security policy. This criteria requires that these additional auditing functions must be able to monitor the occurrence or accumulation of security auditable events that may indicate such violations. Thus, these additional events must be accounted for in the audit trail.

Design:

(C2):

- Establish a mechanism for operation of the Audit Function

- Design a mechanism to monitor security auditable events and to notify security administrator

The design of the auditing functions in a Class B3 TCB must satisfy the requirements stated above. Thus, its design should incorporate that of Classes C2, B1, and B2. In addition, it must include the design of a mechanism to monitor security auditable events and to notify the security administrator accordingly. This is to ensure that the security administrator is kept well informed of such events so that he can take appropriate and prompt action in response to their occurrence.

3.3.3 Assurance

3.3.3.1 Operational Assurance

3.3.3.1.1 System Architecture

(C1):

o TCB shall maintain a domain for its own execution that protects it from external interference or tampering

(B1):

o TCB shall maintain process isolation through the provision of distinct address spaces under its control

(B2):

o TCB shall be internally structured into well-defined largely independent modules
o TCB shall make effective use of available hardware to separate those elements that are protection-critical from those that are not
o TCB modules shall be designed such that the principle of least privilege is enforced
o Features in hardware, such as segmentation, shall be used to support logically distinct storage objects with separate attributes (namely: readable, writable)
o TCB user interface shall be completely defined and all elements of the TCB identified
o TCB shall be designed and structured to use a complete, conceptually simple protection mechanism with precisely defined semantics; this mechanism shall play a central role in enforcing the internal structuring of the TCB and the system
o TCB shall incorporate significant use of layering, abstraction, and data hiding. Significant system engineering shall be directed toward

minimizing the complexity of the TCB and excluding from the TCB modules that are not protection-critical

Requirements:

(B1):

- Identify all resources and distinct address spaces to be protected by the TCB, including the TCB code and data structures

(B2):

- Identify all ways in which a user can interface with the TCB by identifying all elements of the TCB and identifying the user interface to each

Design:

(C2):

- Establish method of isolating resources to be protected by the TCB

(B1):

- Establish a mode for protecting each resource and distinct address space within the TCB

(B2):

- Establish the user interface for the TCB

3.3.3.1.2 System Integrity

(C1):

c Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB

Requirements:

(C1):

- Identify hardware and firmware elements of the TCB to be validated
- Determine validation criteria for testing each element of the TCB

- Determine the appropriate interval at which validation of the hardware firmware should take place

Design:

(C1):

- Establish a method for testing each criterion for each element of the TCB

3.3.3.1.3 Covert Channel Analysis

o System developer shall conduct a thorough search for covert channels and make a determination of the maximum bandwidth of each identified channel

As mentioned in the Covert Channel Analysis section of the Class B2 TCB discussion, this analysis is performed by the system developer and requires no software development. As such, no discussion of this topic is necessary. It is important, however, to note that in the development of a Class B3 TCB, the search performed by the system developer is for all covert channels. The scope of the search is expanded from that of the Class B2 TCB, because covert channels include both covert storage channels and covert timing channels.

3.3.3.1.4 Trusted Facility Management

(B2):

o TCB shall support separate operator and administrator functions
o Functions performed in the role of security administrator shall be identified
o ADP system administrative personnel shall only be able to perform security administrator functions after taking a distinct auditable action to assume the security administrator role on the ADP system
o Non-security functions that can be performed in the security adminstration role shall be limited strictly to those essential to performing the security role effectively

Requirements:

(B2):

- Identify all functions to be performed by the operator and administrator
- <u>Classify administrator functions as system or security functions</u>

The major distinction between Trusted Facility Management in a Class B2 TCB and Trusted Facility Management in a Class B3 TCB deals with the level of protection for the administrative functions. In Class C2, all of these functions, including system administration and security administration, are all at the same level of protection. In a Class B3 TCB, although these functions are all accessible via the administration function, a special auditable action is required to access the security administration functions. To develop the administrator function properly, an additional requirement is necessary. In addition to identifying all functions to be performed by the administrator, each identified function must be classified as to whether it is a system function or a security function.

Design:

(B2):

- Establish a mechanism for the proper operation of each of the functions for the operator and each of the functions for the administrator

3.3.3.1.5 Trusted Recovery

o <u>Procedures and/or mechanisms shall be provided to assure that, after an ADP system failure or other discontinuity, recovery without a protection compromise is obtained</u>

Requirements:

- <u>Identify a means for assuring that system failure will not result in a protection compromise</u>

To properly determine the requirements for the trusted recovery of a Class B3 TCB, a means must be identified for assuring that after an ADP system failure or other discontinuity, system recovery can be achieved without compromising the security protection of the TCB. When such a failure occurs, all accesses to the TCB, especially through communication channels and device I/O, are securely terminated. During system recovery the state of accesses to the TCB must be checked to ensure that no unauthorized access is possible. Thus, the integrity of the security of the TCB must be maintained during system recovery as well as during normal system operation.

Design:

- <u>Establish a mechanism for recovering from system failure in a trusted manner</u>

The design of a mechanism for trusted recovery from an ADP system failure must satisfy the requirements stated above. It must provide a means for the system to completely maintain the integrity of the TCB's security during failure and system recovery so that no unauthorized access (e.g., through covert channels) to the TCB is allowed during these vulnerable times.

3.3.3.2 Life-Cycle Assurance

3.3.3.2.1 Security Testing

(C1):

o Security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation

(B1):

o Team of individuals who thoroughly understand the specific implementation of the TCB shall subject its design documentation, source code, and object code to thorough analysis and testing. The team's objectives will be (1) to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy, and (2) to assure that no subject is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users
o TCB shall be <u>found resistant to</u> penetration

(B2):

o All discovered flaws shall be corrected and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced
o Testing shall demonstrate that the TCB implementation is consistent with the descriptive top-level specification
o <u>No design flaws and no more than a few correctable implementation flaws may be found during testing, and there shall be reasonable confidence that few remain</u>

Requirements:

(C1):

- Identify security protection mechanisms of the TCB to be tested

Design:

(B1):

- Establish means of testing each security protection mechanism in a manner consistent with the stringent requirements detailed in the TCSEC

Coding:

(C1):

- Use prudently placed debug statements to allow the tester to monitor operation of the security mechanisms

3.3.3.2.2 Design Specification and Verification

(B2):

- o Formal model of the security policy supported by the TCB shall be maintained that is proven to be consistent with its axioms
- o A descriptive top-level specification (DTLS) of the TCB shall be maintained that completely and accurately describes the TCB in terms of exceptions, error messages, and effects. It shall be shown to be an accurate description of the TCB interface.
- o <u>A convincing argument shall be given that the DTLS is consistent with the model</u>

Design:

(B1):

- Identify the model to be utilized
- Prove consistency between design and the model

3.3.3.2.3 Configuration Management

(B2):

- o A configuration management system shall be in place that maintains control of changes to the descriptive top-level specification, other design data, implementation documentation, source code, the running version of the object code, and test fixtures and documentation
- o The configuration management system shall assure a consistent mapping among all documentation and code associated with the current version of the TCB
- o Tools shall be provided for generation of a new version of the TCB from source code and for comparing a newly generated version with the previous TCB version in order to ascertain that only the intended changes have been made in the code that will actually be used as the new version of the TCB

3.3.4 Documentation

(C1):

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for the TCB. The documentation must be developed to meet all requirements set forth in the TCSEC, and must be complete and up-to-date; however, this is not particular to this development and requires no further discussion.

3.3.4.1 Security Features User's Guide

(C1):

- o Single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another

3.3.4.2 Trusted Facility Manual

(C1):

- o Manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility

(C2):

- o The procedures for examining and maintaining the audit files, as well as the detailed audit record structure for each type of audit event, shall be given

(B1):

- o Manual shall describe the operator and administrator functions related to security, to include changing the security characteristics of a user
- o Manual shall provide guidelines on the consistent and effective use of the protection features of the system, how they interact, how to securely generate a new TCB, and facility procedures, warnings, and privileges that need to be controlled in order to operate the facility in a secure manner

(B2):

- o TCB modules that contain the reference validation mechanism shall be identified
- o Procedures for secure generation of a new TCB from source after modification of any modules in the TCB shall be described
- o <u>Manual shall include the procedures to ensure that the system is initially started in a secure manner</u>
- o <u>Procedures shall also be included to resume secure system operation after any lapse in system operation</u>

3.3.4.3 Test Documentation

(C1):

- o System developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing

(B2):

- o Documentation shall include results of testing of effectiveness of the methods used to reduce covert channel bandwidths

3.3.4.4 Design Documentation

(C1):

- o Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB

(B2):

- o The interfaces between the TCB modules shall be described
- o A formal description of the security policy model enforced by the TCB shall be available and proven to be sufficient to enforce the security policy
- o Specific TCB protection mechanisms shall be identified and an explanation given to show that they satisfy the model
- o The DTLS shall be shown to be an accurate description of the TCB interface
- o TCB implementation (i.e., hardware, firmware, and software) shall be informally shown to be consistent with the DTLS
- o The elements of the DTLS shall be shown, using informal techniques, to correspond to the elements of the TCB
- o Documentation shall describe how the TCB implements the reference monitor concept and give an explanation why it is tamper resistant, cannot be bypassed, and is correctly implemented
- o Documentation shall describe how the TCB is structured to facilitate testing and to enforce least privilege
- o Documentation shall also present the results of covert channel analysis and the tradeoffs involved in restricting the channels
- o All auditable events that may be used in the exploitation of known covert storage channels shall be identified
- o The bandwidths of known covert storage channels, the use of which is not detectable by the auditing mechanisms, shall be provided

4.0 REFERENCES

Department of Defense Trusted Computer System Evaluation Criteria.
National Computer Security Center, December 1985

Final Evaluation Report of Scomp. Secure Communications
Processor STOP Release 2.1, 23 September 1985

Trusted Computer System Security Requirements Guide for DoD Applications.
1 September 1987

Trusted Network Interpretation of the Trusted Computer System
Evaluation Criteria. National Computer Security Center,
31 July 1987

APPENDIX B

Benefits of and Potential Deterrents to Using Ada in the Software Development Process of Trusted Computing Bases

Prepared for:

National Computer Security Center
9800 Savage Road
Fort Meade, MD 20755

Prepared by:

Ada Applications and
Software Technology Group

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

April 1989

APPENDIX B

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Background

One intent of this appendix is to identify benefits of using Ada in the software development of TCB systems. These benefits include Ada's assets in the application of sound software engineering principles, such as data abstraction, information hiding, modularity, and localization. Also included among the benefits are such Ada constructs as strong data typing, packages, subprograms, and tasks. Also the benefits of using tools in the develpment of Ada language software are discussed.

Another intent of this appendix is to identify and categorize potential deterrents of using Ada in the software development of TCB systems. These include shortcomings inherent to programming languages in general; shortcomings unique to the Ada language.

This appendix is meant to stand alone; however, a familiarity with the TCSEC is recommended to be better able to use this appendix. Also, familiarity with the Ada programming language is helpful.

1.2 Format

This appendix consists of four sections. The first section is an introduction which consists of background information and definitions of key terms that appear in this report. The key terms section includes terms found in the glossary of the TCSEC, the Reference Manual for the Ada Programming language (LRM) [ANSI/MIL-STD-1815A-1983], and the IEEE Standard Glossary of Software Engineering Terminology [IEEE 1983]. Sections 2.0 and 3.0 address the benefits and deterrents issues in two ways. First, Section 2.0 focuses on the following issues: (1) Benefits of using Ada in Developing TCB systems, (2) Potential deterrents to using Ada in developing TCB systems, (3) Shortcomings inherent to programming languages in general in developing TCB systems, (4) Shortcomings unique to the Ada language in developing TCB systems, (5) Benefits of using tools for developing Ada software for TCB systems. Section 3.0 presents these issues in the context of a mapping of Ada usage to generalized TCB criteria. In particular, class B3 as defined in the TCSEC is used as a template for the generalized TCB criteria. Ada constructs and features are identified that may be used to implement TCB functions and features. Though potential problems are identified with using various Ada constructs and features, this is not meant to imply that any of the constructs and features should not be used; only that the security of the TCB must not be compromised when any of the constructs and features are used in the implementation of the TCB. Section 4.0 consists of a summary of this report and its conclusions.

1.3 Key terms

Several key words appear throughout the text of this appendix. These words have specific meanings within the context of certified systems, and their definitions are presented here. These definitions are taken directly from the TCSEC:

Access - A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

Audit Trail - A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions.

Covert Channel - A communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Data - Information with a specific physical representation.

Discretionary Access Control - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Mandatory Access Control - A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

Object - A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc.

Sensitivity Label - A piece of information that represents the security level of an object and that describes the sensitivity (e.g., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions.

Subject - An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

Trusted Computing Base (TCB) - The totality of protection mechanisms within a computer system -- including hardware firmware, and software -- the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Additional Terms

These terms are included because they appear frequently in the following text.

Pragma - A compiler directive. That is, it "is used to convey information to the compiler." According to the Ada language reference manual [ANSI/MIL-STD-1815A-1983], the predefined pragmas (Refer to Annex B in this manual for descriptions) "must be supported by every implementation. In addition, an implementation may provide implementation-defined pragmas, which must then be described in Appendix F", i.e., the appendix on implementation-dependent characteristics that the Ada compiler vendor must provide in his Ada language reference manual.

Security - The protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure. Security also pertains to personnel, data, communications, and the physical protection of computer installations [IEEE 1983]. Specifically, for the purposes of this report, security is defined by the criteria in the TCSEC, i.e., a given security problem corresponds with a specific TCSEC criteria.

To better appreciate the viewpoint of this appendix, the reader is advised that the word "may" is used frequently to avoid incorrect absolute blanket assertions. In particular, the use of "may" indicates that potential security risks are subject to arise when using the various Ada (and other higher order language (HOL)) constructs and techniques. That is, the presence and severity of any given security risk depends on how the various constructs and techniques are implemented and the context in which they are used. Therefore, the use of any given construct or technique <u>may</u> compromise security, depending on its implementation and the context of its use.

2.0 BENEFITS AND POTENTIAL DETERRENTS OF USING ADA IN DEVELOPING TCB SYSTEMS

This section addresses three issues: benefits of using Ada, shortcomings of using higher order languages (HOLs) in general and Ada in particular, and the benefits of using system development tools. Although potential problems are identified with using various HOLs and Ada features, this is not meant to imply that any of the features should not be used. Rather, the security of the TCB must not be compromised when any of the constructs and features are used in the implementation of the TCB. Similarly, although currently available system development tools have their limitations, the benefits of using them may offset these limitations, especially when compared with not using any tools.

Though this report does not advocate abstaining from using any Ada constructs, it should be noted that each of them, to varying degrees, contributes to a large Ada runtime library. Eric Anderson, in his paper, "Ada's Suitability for Trusted Computer Systems," proposes the extreme view of minimizing the Ada runtime library for the security kernel as a primary goal itself, because its size may lend itself to hiding covert channels and Trojan horses. This problem of an excessively large Ada runtime library is in addition to the potential deterrents associated with the various Ada constructs that are discussed below.

To minimize the size of the Ada runtime library Eric Anderson proposes the following severe restrictions on the use of Ada constructs in creating the security kernel: not allowing use of any dynamic storage, tasking, exception handling, any Ada standard packages other than STANDARD and SYSTEM; and limiting the use of runtime constraint checking, math and conversion routines, and representation clauses. Addressing the security issues associated with the Ada runtime library are beyond the scope of this report. Clearly, though, Ada runtime library security issues need to be addressed by further research.

2.1 Benefits of Using Ada in Developing TCB Systems

The following beneficial features are available in the Ada programming language.

2.1.1 **Data Abstraction:**

Ada's data abstraction mechanisms are well suited to represent the data objects in a system's design, namely, that of a TCB. They serve the conceptual manipulation of the data objects in a relatively high-level of abstraction without regard to their underlying representation. Ada allows data to be abstracted with abstract data types. An abstract data type is a construct that "denotes a class of objects whose behavior is defined by a set of values and a set of operations" [Booch 1987A, p.613]. Ada's two primary features that promote the creation of abstract data types are its strong data typing facilities and its packaging mechanism. Strong data typing serves to isolate data types. A package can be used to define an abstract data type by encapsulating its underlying data types and the operations associated with the abstract data type. Ada's strong data typing facility allows the creation of user-defined types, namely, subtypes and derived types. Using Ada's data abstraction techniques aids the representation of data objects in the problem space of any system, including a TCB. "Abstraction aids in the maintainability and understandability of systems by reducing the details a developer needs to know at any given level" [Booch 1987B, p.33]. Data abstraction enhances understandability by allowing the review of system design to focus on high-level, abstract data elements instead of smaller component parts. This increases the understandability of both the design and the code. Thus, this sound software engineering technique can promote an Ada TCB system's design, implementation, and maintenance.

2.1.2 **Information Hiding:**

Ada's information hiding facilities complement its data abstraction capabilities. Whereas abstractions extract the essential details of a given level, "the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system" [Ross, Goodenough, Irvine 1975, p.67]. "Information hiding therefore suppresses how an object or operation is implemented, and so focuses our attention on the higher abstraction" [Booch 1987B, p.33]. Two Ada constructs that are well suited for implementing information hiding are packages and private types. "Packages can be used to hide information from the rest of the program while making explicit the interface with other program parts. This has the advantage that implementation details of each package can be changed by altering only its body, and that the rest of the program may be understood without reference to these details" [Nissen and Wallis 1984, p.122]. As much as possible of the implementation detail should be hidden in the body of the package.

Hiding the information about the implementation of the data abstraction of a data object is achieved in Ada by encapsulating the abstraction in a package, i.e., by hiding the implementation of the object and controlling access to the object so as to encourage and enforce the abstraction. This typically is done with the use of private types and limited private types. In particular, Ada's private types enable the focus to be placed on higher-level real-world abstractions rather than on the details of an implementation. The following implicit operations may be performed on private types: assignment, tests of equality and inequality, explicit type conversion, membership tests, type qualification, and the use of selected components for the selection of any of the private type's discriminant. For limited private types, though, only those operations defined in the corresponding package specification are allowed. For more detailed information on private types, refer to the Ada LRM [ANSI/MIL-STD-1815A-1983]. "Private types prevent misuse of structures by users, presenting them only with the abstract operations appropriate for the abstractions involved" [Nissen and Wallis 1984, p.131].

An example of using Ada's information hiding (and data abstraction) would be to implement a package that defined a linked list structure. Only those details required by a user of the linked list package would be provided in the package specification (data abstraction), e.g., the operations allowed on the linked list. The implementation, though, of the linked list would be hidden inside the package body. The user of the package does not need to know how the linked list is implemented, therefore, information on its implementation is hidden from the user.

The understandability of systems is enhanced "when, at each level of abstraction, we permit only certain operations and prevent any operations that violate our logical view of that level" [Booch 1987B, p.33]. Hiding information about implementation details of subprograms in package bodies offers various benefits. These include protecting the integrity of the subprograms from undue alteration by user's of the subprograms. Also it allows a user of the subprograms specified in the package specification to think in a higher level of abstraction rather than being mired in the subprograms' implementation details. Thus, this sound software engineering technique can promote an Ada TCB system's design, implementation, and maintenance.

2.1.3 **Modularity and Localization:**

Modularity provides the mechanism for collecting logically related abstractions. It deals with how the structure of an object can make the attainment of some purpose easier. Modularity is purposeful structuring, which is usually achieved in a large system by decomposing the system top-down with modules that are either functional (procedure-oriented) or declarative (object-oriented) [Booch 1987B, p.34]. It is composed preferably of existing reusable bottom-up software components. This structuring should be performed to minimize the coupling between modules (i.e., minimizing dependencies between modules) and to strengthen the cohesion within modules (i.e., the components of a given module are functionally and logically dependent) [Booch 1987B, p.34].

Localization is the collecting of logically related computational resources in one physical module that is sufficiently independent of other modules. Localization thus helps to create modules that exhibit loose coupling and strong cohesion.

The principles of modularity and localization directly support modifiability, and understandability [Booch 1987B, p.34]. Any given module should be understandable and relatively independent of other modules. Design decisions localized in given modules limit the effects of a modification to a small set of modules. This directly supports TCB development in two ways. First, all the code related to a particular purpose, e.g., discretionary access control, is localized which aids in the understanding the implementation. Also, a change in the implmentation should not propogate beyond the local modules. Thus, the use of modularization that limits the interconnections among program modules, and the localizing of logically related resources into modules are sound software engineering techniques that can promote an Ada TCB system's design, implementation, and maintenance.

2.1.4 Dynamic Storage with Access Types:

Dynamic storage is a pool of memory, or heap, that is used for storing data whose demand for memory varies during program execution. Despite the problems associated with dynamic storage and Ada's access types discussed below, it is a useful mechanism that can provide a convenient and flexible means of managing memory when the need for memory is constantly changing. Though Ada also provides an automatic garbage collection facility, this management of dynamic storage may result in data being left in the memory heap when the memory is deallocated.

The security of a TCB's dynamically stored data that is about to be collected by this garbage collection can be protected by deleting the data just before it is collected as garbage. That is, sensitive data that may be accessed by an unauthorized user or subject must be removed from memory before the memory is deallocated. Also memory should be scrubbed just before it is dynamically allocated with Ada's **new** statement. Note that these additional checks will impinge on system performance. It would be desirable to include in the next revision of Ada (Ada9x) a predefined pragma that directs Ada compilation to include code to scrub dynamically allocated and deallocated memory after acquiring it from and returning it to the heap respectively.

2.1.5 **Concurrent Programming with Tasking:**
In contrast to other languages Ada incorporates its concurrent programming mechanism, namely, tasking, as an integral part of its definition. Concurrent processes, in particular, tasks, are processes that may execute in parallel on multiple processors or independently scheduled processes on a single processor. That is, they involve the simultaneous, or timeshared, execution of processes. Tasks may interact with each other, and one task may suspend execution pending receipt of information from another task or the occurrence of an external event. Despite the problems associated with concurrent programming and Ada's tasking discussed below, it is a useful mechanism that likely is required for the effective implementation of a TCB system. Safeguards, as required by a TCB's class requirements, in communication between tasks must be enforced in the TCB system, e.g., with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). Communications between tasks (e.g., rendezvous) should be logged in the audit trail.

2.1.6 **Compilation of Specifications:**
Ada provides the ability to compile package and subprogram specifications during the design stage of system development. This aids in the early checking of interfaces between packages, and with parameter consistency between subprograms in the same package. The data structures are declared in the specifications, and consistent use of data types can be automatically checked during design. The use of Ada's capability to compile package and subprogram specifications, therefore, allows early checking of a system's desing and interfaces before the designers and implementors get mired in implementation details. Thus, the quality of the TCB is promoted by using specification compilation.

2.1.7 **Reusable Code:**
The implementation of an Ada TCB system can be aided with the reuse of evaluated code that has been demonstrated to sufficiently satisfy the

security class of the given TCB. Ada's generic units are helpful in creating reusable code. "Generics provide a powerful means by which a program may be 'factorized' in order to shorten code, and reduce incidence of errors, by avoiding redefining items which appear in several places in the program" [Nissen and Wallis 1984, p.181]. When code is reused the number of errors in the code is reduced because errors in such code are fixed when they are identified. Additional time and effort is required during the design phase of developing reusable code for a TCB system. The additional time and effort will pay for itself when the resulting reusable code is used in multiple instances in the current TCB and future TCBs. Thus, the reuse of evaluated code is a sound software engineering technique that can promote the efficient production of an Ada TCB system's design, implementation, and maintenance.

2.1.8 **Standardization and Portability:**
Because Ada is standardized (by the Department of Defense (DoD)), no subsets or supersets of Ada are allowed. This standardization is assured by the DoD's process of validation of Ada compilers. Before any compiler can be used on a DoD contract, it must be validated by passing a series of tests to ensure that it implements the LRM [ANSI/MIL-STD-1815A-1983] precisely. This aids the portability of Ada software among different types of computers. In particular, an Ada TCB system is more likely to be easier to port between two different types of computers than if the TCB were developed in another language.

2.2 Potential Deterrents to Using Ada in Developing TCB Systems

2.2.1 Shortcomings Inherent to Programming Languages in General in Developing TCB Systems

The following features are typically available in programming languages. Their use poses potential deterrents to the security of TCB systems.

2.2.1.1 **Static Storage:**
Static storage, such as local variables and arrays, are typically provided in programming languages. When a portion of static storage is no longer needed during the execution of a program, it may still contain data to which unauthorized users or subjects should not have access. Therefore, such static storage, that is no longer needed, should be scrubbed.

2.2.1.2 Dynamic Storage:
Dynamic storage, as discussed above, is a convenient way of managing a given system's continually varying demands for memory. Dynamic storage is typically used to implement such constructs as linked lists and queues. This introduces the possibility that sensitive data may be accessed by an unauthorized user or subject when it has not been removed from newly allocated or deallocated memory.

2.2.1.3 **Input and Output:**
For any TCB that has input and output capabilities, it must not allow an unauthorized user or subject to gain access to the TCB and its data. For example, users must be prevented from accessing disk files or tapes that they are not authorized to access. Also, the use of security sensitive terminals must be restricted to those users who have authorization to use the terminals. The security of the TCB's input and output functions should be provided with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). All inputs to and outputs from a TCB should be logged in the audit trail.

2.2.1.4 Global Variables:
Global variables (or partially global variables, e.g., in FORTRAN COMMON blocks) are a convenient means of passing data between different parts of a system. The use of global variables causes difficulty in tracing modifications to the variables. Thus, the security of the data is more difficult to regulate in a TCB.

2.2.1.5 Concurrent Processing:
As discussed above, concurrent processing involves the simultaneous, or timeshared, execution of processes. The communications between the concurrent processes may allow the introduction of covert channels. If multiple processes are executing concurrently, then one process could detect the extent of demands being placed on system resources by the system's responsiveness to the process's demands. For example, a process might be able to execute a series of subprograms in a given amount of time when it is the only process executing. The amount of time to execute the same set of subprograms might be much greater when several processes are executing. By monitoring the time required to execute its own subprograms, this process can assess the relative system load. Although the seriousness of this type of covert channel is heightened by concurrent processing, the increase in performance warrants that concurrence be used with caution rather than be totally avoided. The security of communications between concurrent processes should be managed with appropriate discretionary access controls

(e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). Communications between tasks (e.g., rendezvous) should be logged in the audit trail.

2.2.1.6 **Go To Statements:**
Go To statements provide the means for a program to transfer control, in an unstructured manner, of its operation elsewhere in the program. The larger the module, the more potential problems that Go To statements can present. In particular, undesired redirection of program execution may be inserted and be hard to track. Because this transfer of control may be unregulated, ensuring the security of a TCB system is very difficult. Also, most HOLs, such as Ada, provide more structured control structures (e.g., for and while loops) that make using Go To statements unnecessary.

Example: This example illustrates unstructured transfer of control of program execution by Go To statements.

```
<< Backward_Label >>

-- . . . Many lines of code!

goto Forward_Label;

-- . . .

goto Backward_Label;

-- . . . Many lines of code!

<< Forward_Label >>
```

2.2.1.7 **Interrupts:**
Interrupts provide asynchronous means of altering program execution so that an external event can be handled by the system. This may allow the handling of an external event so that an unauthorized user or subject is able to gain access to a TCB and its data. Interrupts should be managed with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). Interrupts should be monitored by logging them in the audit trail.

2.2.1.8 Lack of Modularity:

A primary advantage to modular systems is increased understandability; the primary problem with a lack of modularity is a lack of understandability. Failure to adequately modularize a TCB's implementation introduces several problems in its development and maintenance. In particular, lack of modularity causes the system's design and implementation to be less understandable because the high-level abstraction aspects of the system are obscured by the exorbitant amount of details in large modules. This lack of understandability has an adverse effect on testing, because not all of the module's functions and subfunctions can be easily identified and then tested. Also, lack of modularity complicates testing because the typically greater number of possible logical paths (i.e., transfers of control of program execution) in large modules cause them to be more difficult to test thoroughly than small modules, which have few possible logical paths. This testing is further complicated by the use of Go To statements in large modules. A lack of modularity has a great adverse effect on maintainability also. Because a large module cannot be as well understood as a smaller module, any ripple effect of a change to the module will be very difficult to detect and control.

2.2.2 Shortcomings Unique to the Ada Language in Developing TCB Systems

This section details how particular constructs and features that may be detrimental to the development of TCB systems are implemented in Ada. Although these constructs and features introduce the potential for circumventing security attributes, their use may at times be necessary. Also, of concern is the effect when multiple features in this list are used together. Such interactions are discussed below with appropriate constructs and features. These concerns will be addressed in the programming guidelines.

2.2.2.1 Static Storage:

Static storage is a portion of memory that is set aside at compile time and whose size does not change during program execution. Objects of most Ada data types use static storage. Statically stored data, like that in an array, can generally be accessed more efficiently than data stored dynamically in data structures consisting of access types. This is offset by their potentially inefficient use of memory; this is particularly true of arrays. It would be desirable to include in the next revision of Ada (Ada9X) a predefined pragma that directs Ada compilation to include code that scrubs memory that is local to a subprogram or package just before the scope of the subprogram or package is left during program execution, i.e., just before the visibility of the static (or dynamic) memory is lost. These conflicting needs, efficient execution versus efficient use of memory, are particularly important for large data structures.

2.2.2.2 **Dynamic Storage with Access Types:**
Access types are used in Ada to implement dynamic storage constructs, like linked lists and queues. Thus Ada also is subject to the same problems with securely handling dynamic storage that are associated with other languages. "An implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible" [ANSI/MIL-STD-1815A-1983].

In addition, Ada has the **pragma** CONTROLLED, which "specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program" [ANSI/MIL-STD-1815A-1983]. The use of this pragma may allow unauthorized access to dynamic storage that is yet to be deallocated. Also, "if an object or one of its subcomponents belongs to a task type, it is considered to be accessible as long as the task is not terminated". This presents the possibility that unauthorized access to dynamic storage may be available between tasks.

2.2.2.3 **Actication Records**
When program execuition leaves the scope of a library unit, the memory containing its data is returned to the heap without being scrubbed.

2.2.2.4 Global Variables:
"Global variables" may be implemented in Ada either in a package specification or a subprogram specification or used as shared variables associated with tasks (which are established with the pragma SHARED). The primary advantage of using global and shared variables is that some efficiency in data transfer within the program is typically introduced. This advantage is almost always offset by the problems caused by using global and shared variables. Using global variables in Ada poses problems similar to using them in other languages, as discussed above. Ensuring the security of data "shared" among tasks is particularly difficult because the flow of program control is harder to trace when tasking is involved. Therefore, the security of an Ada TCB system is complicated when it incorporates global or shared variables.

Example: This example illustrates global variable scope/visibility deterrent in a package specification. That is, any package or subprogram that withs package Highly_Visible_Package can access and thus modify Global_ACL_Variables in a manner that is difficult to trace.

```
with Access_Control_List_Types_Package;
package Highly_Visible_Package is

  Global_ACL_Variables : Access_Control_List_Types_Package.
                              Named_Object_Record_Type;

  -- . . .

end Highly_Visible_Package;
```

2.2.2.5 **Concurrent Programming with Tasking:**

Tasking is Ada's means of implementing concurrent programming. Tasks and entries have three attributes as specified in the LRM: T'CALLABLE, T'TERMINATED, and E'COUNT. The use of these dynamic attributes enables the passing of information in a manner that is difficult to comprehend. Their use may, therefore, open covert channels. The communications between tasks may also allow the introduction of covert channels. In addition, a tasking implementation may introduce the problems with dynamic storage and global variables discussed above.

Shared variables are a convenient means of passing data between different tasks in a system. Shared variables, associated with tasks, are established with the **pragma** SHARED. The primary advantage of using shared variables is that some efficiency in data transfer within the program is typically introduced. This advantage is almost always offset by the problems caused by using shared variables. Ensuring the security of data "shared" among tasks is particularly difficult, because the flow of program control is harder to trace when tasking is involved. Therefore, the security of an Ada TCB system is complicated when it incorporates shared variables. When a task is terminated the memory containing its data is returned to the heap without being scrubbed.

Example: This example illustrates shared variable scope/visibility deterrent in a **package** specification. That is, any tas' in any **package** or **subprogram** that **withs package** Highly_Visible_ Package can access and thus modify Shared_ACL_Variables i nner that is difficult to trace.

```
with Access_Control_List_Types_Package;
package Highly_Visible_Tasks_Package is
```

```
     Shared_ACL_Variables : Access_Control_List_Types_Package.
                              Named_Object_Record_Type;


    -- pragma SHARED is not currently supported in VAX Ada.
    pragma SHARED ( Shared_ACL_Variables );

      -- Both Resource_A_Task and Resource_B_Task and any other task in
      -- another package that withs package Highly_Visible_Package has
      -- access to Shared_ACL_Variables!

      task Resource_A_Task is
        entry Get_from_Resource_B;
        entry Send_to_Resource_B;
      end Resource_A_Task;

      task Resource_B_Task is
        entry Get_from_Resource_A;
        entry Send_to_Resource_A;
      end Resource_B_Task;

      -- . . .

   end Highly_Visible_Tasks_Package;
```

2.2.2.6 **Use Clauses and Renaming Declarations:**
Ada's "use clause achieves direct visibility of declarations that appear in the visible parts of named packages." Ada's "renaming declaration declares another name for an entity." The use of the use clause and renaming make identifying the origin of an invoked subprogram difficult. The impact on testing is that the specification and definition of the invoked subprogram are not accessible to a tester. Also, a similar effect, which is equally adverse, occurs in maintenance, when it is difficult to identify the origin of the invoked subprogram.

Examples: These examples illustrate the obfuscation introduced by the use clause and renaming declarations. The examples of the use clause and renaming declaration follow the initial declaration of package Generic_Access_Control_List_Manager_Package, which is a trusted library package.

```
   generic

     -- . . .
```

```
  type Name_Type is private;
  type ACL_Record_Type is private;

  -- . . .

package Generic_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in     Name_Type;
        ACL_Record :    out ACL_Record_Type );

    -- . . .

end Generic_Access_Control_List_Manager_Package;


with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;
with Mandatory_Access_Control_Manager_Package;
with Audit_Trail_Manager_Package;
package body Generic_Access_Control_List_Manager_Package is

    -- The user can gain only indirect access to instantiated
    -- access control list through the subprograms declared in the
    -- package specification.  Thus the access control list data
    -- structure is hidden from the user of this package.  The
    -- typical list manipulation operations ( e.g., as illustrated
    -- by Booch 1987A and Feldman 1985 ) are only provided in the
    -- package body.

    -- . . .

    -- Typical list manipulation operations

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in     Name_Type;
        ACL_Record :    out ACL_Record_Type ) is

        -- . . .
        begin  -- Get_ACL_Record

          -- . . .

          -- Sequence through the access control list data structure
          -- using the typical list manipulation operations to locate
          -- and get the indicated access control list record.

          -- . . .

      end Get_ACL_Record;

      -- . . .

   end Generic_Access_Control_List_Manager_Package;
```

Example 1: Obfuscation introduced by the use clause

```
   with Basic_TCB_Types_Package;
   use  Basic_TCB_Types_Package;
   with Access_Control_List_Types_Package;
   use  Access_Control_List_Types_Package;
   with Mandatory_Access_Control_Types_Package;
   with Generic_Access_Control_List_Manager_Package;
   generic

     -- . . .

     type Password_Type is private;

     -- . . .

   package Generic_User_Identification_and_Authentication_Package is

     -- Instantiation of Generic_Access_Control_List_Manager_Package
     package Named_Objects_Access_Control_List_Manager_Package is new
             Generic_Access_Control_List_Manager_Package
               ( Name_Type       => Name_of_Object_Type,
                 ACL_Record_Type => Named_Object_Record_Type );

     use Named_Objects_Access_Control_List_Manager_Package;

     -- The locations of the declarations of Name_of_Object_Type
     -- and Named_Object_Record_Type are now lost due to the
```

```
  -- use clauses, use Basic_TCB_Types_Package and
  -- use Access_Control_List_Types_Package, respectively!

  -- . . .

  procedure Check_Password
      ( Password          : in Password_Type;
        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.
            MAC_Record_Type );

  -- . . .

end Generic_User_Identification_and_Authentication_Package;


with Audit_Trail_Manager_Package;
package body Generic_User_Identification_and_Authentication_Package is

  -- . . .

  procedure Check_Password
      ( Password          : in Password_Type;
        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.
            MAC_Record_Type ) is

      -- . . .

      Name       : Name_of_Object_Type;
      ACL_Record : Named_Object_Record_Type;

      -- . . .

    begin  -- Check_Password

      -- The location of the specification of Get_ACL_Record
      -- is now lost due to the use clause,
      -- use Named_Objects_Access_Control_List_Manager_Package!

      Get_ACL_Record ( Name, ACL_Record );

  end Check_Password;

  -- . . .
```

```
end Generic_User_Identification_and_Authentication_Package;
```

Example 2: Obfuscation introduced by renaming declarations

```
with Basic_TCB_Types_Package;
with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;
with Generic_Access_Control_List_Manager_Package;
generic

  -- . . .

  type Password_Type is private;

  -- . . .

package User_Identification_and_Authentication_Package is

  --  . . .

  procedure Check_Password
      ( Password         : in Password_Type;
        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.
            MAC_Record_Type );

  --  . . .

end User_Identification_and_Authentication_Package;


with Audit_Trail_Manager_Package;
package body User_Identification_and_Authentication_Package is

  -- Instantiation of Generic_Access_Control_List_Manager_Package
  package Named_Objects_Access_Control_List_Manager_Package is new
          Generic_Access_Control_List_Manager_Package
            ( Name_Type       => Basic_TCB_Types_Package.
                                   Name_of_Object_Type,
              ACL_Record_Type => Access_Control_List_Types_Package.
                                   Named_Object_Record_Type );

  --  . . .
```

```
procedure Get_Access_Control_List_Record;
            ( Name       : in     Basic_TCB_Types_Package.
                                    Name_of_Object_Type;
              ACL_Record :     out Access_Control_List_Types_Package.
                                    Named_Object_Record_Type )
  renames Named_Objects_Access_Control_List_Manager_Package.
            Get_ACL_Record;

-- . . .

procedure Check_Password
            ( Password         : in Password_Type;
              Local_MAC_Record : in
                Mandatory_Access_Control_Types_Package.
                  MAC_Record_Type ) is

    -- . . .

    Name       : Basic_TCB_Types_Package. Name_of_Object_Type;
    ACL_Record : Access_Control_List_Types_Package.
                   Named_Object_Record_Type,
    -- . . .

  begin  -- Check_Password

    -- The location of the specification of
    -- Get_Access_Control_List_Record,
    -- i.e., Named_Objects_Access_Control_List_Manager_Package.
    -- Get_ACL_Record is now lost due to the renaming declaration!

    Get_Access_Control_List_Record ( Name, ACL_Record );

end Check_Password;

-- . . .

end User_Identification_and_Authentication_Package;
```

2.2.2.7 Unchecked Programming:

Ada provides two unchecked programming features, unchecked storage deallocation and unchecked type conversion. "The predefined generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are used for unchecked storage deallocation and for unchecked type conversions"

[ANSI/MIL-STD-1815A-1983]. Some Ada compilers, such as the DEC VAX Ada, allow unchecked conversion with private and limited private types. This allows the benefits of information hiding provided by theses private types to be circumvented and thus allows the introduction of security breaching devices, such as covert channels. The use of unchecked storage deallocation may allow an unauthorized user or subject to gain access to the storage. The potential problems associated with its use are similar to those associated with the use of dynamic storage. Using unchecked type conversion can defeat the strong typing capabilities of Ada. This impedes testing the security of an Ada TCB.

Example: This example illustrates the deterrent introduced by unchecked type conversion of a private type, in particular, in VAX Ada. The violations of the private type, Key_Type, are indicated by the inline comments in the procedure Test_Key.

```
package Key_Manager_Package is
    type Key_Type is private;
    Null_Key : constant Key_Type;
    procedure Get_Key ( K : out Key_Type );
    function "<" ( X, Y : Key_Type ) return BOOLEAN;
    function "+" ( X, Y : Key_Type ) return Key_Type;

  private
    type Key_Type is new NATURAL;
    Null_Key : constant Key_Type := 0;
end Key_Manager_Package;


package body Key_Manager_Package is

    Last_Key : Key_Type := 0;

    procedure Get_Key ( K : out Key_Type ) is
      begin
        Last_Key := Last_Key + 1;
        K := Last_Key;
    end Get_Key;

    function "<" ( X, Y : Key_Type ) return BOOLEAN is
      begin
        return NATURAL ( X ) < NATURAL ( Y );
    end "<";

    function "+" ( X, Y : Key_Type ) return Key_Type is
```

```
      begin
        return Key_Type ( NATURAL ( X ) + NATURAL ( Y ) );
    end "+";

end Key_Manager_Package;


with Key_Manager_Package;
with UNCHECKED_CONVERSION;
procedure Test_Key is

    function Key_Content is new UNCHECKED_CONVERSION
      ( SOURCE => Key_Manager_Package. Key_Type,
        TARGET => INTEGER );

    function Tampered_with_Key_Content is new UNCHECKED_CONVERSION
      ( SOURCE => INTEGER,
        TARGET => Key_Manager_Package. Key_Type );

    Key_Value : Key_Manager_Package. Key_Type
                   := Key_Manager_Package. Null_Key;

    Key_Integer  : INTEGER := 7;
    Key_Relation : BOOLEAN := FALSE;

  begin  -- Test_Key

    -- 1.  Test unchecked conversions.

    --   1.a  The following statement illustrates unchecked conversion
    --        of an object of INTEGER type and its assignment to
    --        Key_Value, which is a private type!

    Key_Manager_Package. Get_Key ( Key_Value );
    Key_Integer := Key_Content ( S => Key_Value );


    --   1.b  The following statement illustrates unchecked conversion
    --        of an object of INTEGER type and its assignment to
    --        Key_Value, which is a private type!

    Key_Integer := 13;
    Key_Value   := Tampered_with_Key_Content ( S => Key_Integer );
```

```
-- 2.  Test expressior compatibility with unchecked conversions.

--    2.a  Test expression compatibility of an object of
--         Key_Manager_Package.Key_Type and an object of INTEGER
--         type.

Key_Integer := 10;
Key_Manager_Package. Get_Key ( Key_Value );
Key_Integer := Key_Integer + Key_Content ( S => Key_Value );


--    2.b  Test expression compatibility of an object of INTEGER
--         type and an object of Key_Manager_Package.Key_Type.

Key_Manager_Package. Get_Key ( Key_Value );
Key_Integer := 20;

declare
    function "+" ( X, Y : Key_Manager_Package. Key_Type ) return
      Key_Manager_Package. Key_Type renames
        Key_Manager_Package. "+";

  begin

    Key_Value := Key_Value +
                 Tampered_with_Key_Content ( S => Key_Integer );
end;  -- block


-- 3.  Test parameter compatibility with unchecked conversions.

--    3.a  Test parameter compatibility of an object of INTEGER
--         type to an object of Key_Manager_Package.Key_Type.

Key_Integer := 30;
Key_Manager_Package. Get_Key ( Key_Value );
Key_Integer := Key_Integer * Key_Content ( S => Key_Value );


--    3.b  Test parameter compatibility of an object of
--         Key_Manager_Package.Key_Type to an object of INTEGER
--         type.

Key_Relation := FALSE;
```

```
    Key_Manager_Package. Get_Key ( Key_Value );
    Key_Integer  := 40;
    Key_Relation :=
      Key_Manager_Package. "<" ( Key_Value,
            Tampered_with_Key_Content ( S => Key_Integer ) );
  end Test_Key;
```

2.2.2.8 **Exceptions:**

The major difficulty with exceptions [Tripathi] in the Ada language from the point of view of software development of TCBs is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during testing of programs. This complexity is furthered by the fact that exceptions are propagated "as is," which could cause an unhandled exception to propagate from several levels down to a routine that has no understanding of the meaning of the exception. A stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions - conversion of unhandled exceptions to some ROUTINE_ERROR on exit from a block (within a package or not), or explicit use of **others** clauses at all possible functions (not a convenient approach) - the complexity could be reduced.

Another matter of concern with respect to exceptions is due to the non-specificity of the language with respect to modes of parameter passing. If a compiler passes an **in out** parameter by copy on entry and on exit, the actual parameter may never be updated if the routine raises an exception, whereas if the parameter is passed by reference, changes to the actual parameter may change the passed formal parameter, and the value will have been updated in the presence of a raised exception.

In addition using **pragma** SUPPRESS prevents the raising of exceptions for selected checks, which can serve to monitor the proper execution of the program during runtime. Thus, this pragma also can adversely affect the TCB security. It should also be noted that Ada code generated when using **pragma** SUPPRESS cannot be trusted to work as expected, because Ada compilers currently are not validated when **pragma** SUPPRESS is used.

Examples: These examples illustrate the deterrent introduced by transfer of control of program execution by exception handling. In the first example, an exception raised during execution of a deeply-nested subprogram can cause control flow to be unpredictably altered. In the second example, an exception is raised during execution of a task which is handled by another task, thus, control flow also can be unpredictably altered.

Example 1: Deterrent introduced by transfer of control of program execution by exception handling through several nested subprograms

```
package Handle_Nested_Subprograms_Exceptions_Package is

  procedure Handle_Exceptions;

end Handle_Nested_Subprograms_Exceptions_Package;


with TEXT_IO;
package body Handle_Nested_Subprograms_Exceptions_Package is

  Password_is_Invalid : exception;

  procedure Check_Password is

    Password_is_Incorrect : BOOLEAN := FALSE;

    begin  -- Check_Password

      -- . . .

      if Password_is_Incorrect then

        raise Password_is_Invalid;

      end if;

      -- . . .

  end Check_Password;

  procedure Handle_Exceptions is

    begin  -- Handle_Exceptions

      -- . . .

      Check_Password;
      -- . . .

    exception

      -- . . .

      when Password_is_Invalid =>
             TEXT_IO. PUT ( "The password is invalid!" );

      -- . . .

      when others =>
             TEXT_IO. PUT ( "We have a problem here!" );

    end Handle_Exceptions;

  end Handle_Nested_Subprograms_Exceptions_Package;
```

Example 2: Deterrent introduced by transfer of control of program execution by exception handling across tasks

```
  package Handle_Exceptions_Across_Tasks_Package is

    procedure Check_Password;

  end Handle_Exceptions_Across_Tasks_Package;


  with TEXT_IO;
  package body Handle_Exceptions_Across_Tasks_Package is

    Password_is_Invalid : exception;

    task Raise_Invalid_Password_Exception_Task is

      -- . . .

      entry Check_Password_Validity;

      -- . . .

    end Raise_Invalid_Password_Exception_Task;
```

```
task Handle_Exceptions_Task is

  -- . . .

  entry Handle_Invalid_Password;

  -- . . .

end Handle_Exceptions_Task;

procedure Check_Password is

  begin  -- Check_Password

    -- . . .

    Raise_Invalid_Password_Exception_Task. Check_Password_Validity;

    -- . . .

    Handle_Exceptions_Task. Handle_Invalid_Password;

    -- . . .

end Check_Password;

task body Raise_Invalid_Password_Exception_Task is

  Password_is_Incorrect : BOOLEAN := FALSE;

  begin  -- Raise_Invalid_Password_Exceptions_Task

    -- . . .

    accept Check_Password_Validity do

      -- . . .

      if Password_is_Incorrect then

        raise Password_is_Invalid;
```

```
          end if;

          -- . . .

        end Check_Password_Validity;

        -- . . .

    end Raise_Invalid_Password_Exception_Task;

    task body Handle_Exceptions_Task is

      begin  -- Handle_Exceptions_Task

        -- . . .

        accept Handle_Invalid_Password;

        -- . . .

      exception

        -- . . .

        when Password_is_Invalid =>
               TEXT_IO. PUT_LINE ( "The password is invalid!" );

        -- . . .

        when others =>
               TEXT_IO. PUT_LINE ( "We have a problem here!" );

    end Handle_Exceptions_Task;

    -- . . .

  end Handle_Exceptions_Across_Tasks_Package;
```

2.2.2.9 Interface with other languages:
Ada provides the means to interface with other languages using the **pragma** INTERFACE. Using multiple high-order languages, or using assembly language with a high-order language, makes a system more difficult to understand. The difficulty here is not with the **pragma** INTERFACE per se, but rather with

using more than one language in a system. Ada code has more potential of being self-documenting than other high-order languages because its syntax is logical and similar to English. Also, Ada allows the use of a sufficient number of characters in its identifiers to make them more readable and understandable than the identifiers of other high-order languages.

The difficulty with allowing machine code insertions in developing TCBs is the inability to correlate the specification of the machine code instructions with the intended abstract behavior at the Ada language level. If it is possible to specify the intended behavior, it would likely be preferable to program in Ada; if not, attempting to use such insertions would stymie the development of TCBs.

Example 1: This example illustrates interfacing Ada with the programming language C.

```
pragma INTERFACE ( C, rm );  -- obviously, the reference monitor
```

Example 2: This example illustrates interfacing Ada with DEC VAX-11's MACRO assembly language.

```
pragma INTERFACE ( MACRO, TCB );  -- only an Ada shell is required
```

2.2.2.10 **Representation Clauses and Implementation-Dependent Features:**
Ada provides means to directly interact with the underlying hardware and operating system using its representation clauses and implementation-dependent features. As with interfacing with other languages, the beneficial Ada features are no longer available in these parts of the system. Using representation clauses and implementation-dependent features may allow penetration of the TCB through the underlying hardware or operating system, which clearly could allow compromising the security of an Ada TCB system.

Example: This example illustrates the deterrent introduced by an interrupt defined by an address clause. Note that address clauses are not currently supported in VAX Ada.

```
procedure Penetrate_Memory_Space is

   Char : CHARACTER;
```

```
task Input_Penetrator_Task is

  pragma P  ORITY ( 4 );
       -- must have at least the priority of the interrupt

  entry Get_Character_from_Penetrator_Input_Address
          ( Char : out CHARACTER );

  entry Save_Hardware_Buffer_Character;

  -- assuming that SYSTEM. ADDRESS is an INTEGER type
  for Save_Hardware_Buffer_Character use at 16#0020#;

end Input_Penetrator_Task;


task Output_Penetrator_Task is

  pragma PRIORITY ( 4 );
       -- must have at least the priority of the interrupt

  entry Deposit_Character_into_Hardware_Buffer;

  entry Put_Character_into_Penetrator_Output_Address
          ( Char : in CHARACTER );

  -- assuming that SYSTEM. ADDRESS is an INTEGER type
  for Deposit_Character_into_Hardware_Buffer use at 16#0024#;

end Output_Penetrator_Task;


task body Input_Penetrator_Task is

    Max_Size_of_Internal_Input_Buffer : constant POSITIVE
      := 64;

    Internal_Input_Buffer :
      array ( 1 .. Max_Size_of_Internal_Input_Buffer )
        of CHARACTER;

    Input_Buffer_Pointer  : POSITIVE := 1;
    Output_Buffer_Pointer : POSITIVE := 1;

    Buffer_Count : INTEGER := 1;
    Hardware_Character_Buffer : CHARACTER;
    for Hardware_Character_Buffer use at 16#0100#;

  begin  -- Input_Penetrator_Task
    loop
      select
          when Buffer_Count > 0 =>

            accept Get_Character_from_Penetrator_Input_Address
                     ( Char : out CHARACTER ) do

              Char := Internal_Input_Buffer (
                        Output_Buffer_Pointer );

            end Get_Character_from_Penetrator_Input_Address;

            Output_Buffer_Pointer :=
              Output_Buffer_Pointer mod
                Max_Size_of_Internal_Input_Buffer + 1;

            Buffer_Count := Buffer_Count - 1;

        or
          when Buffer_Count <
                 Max_Size_of_Internal_Input_Buffer =>

            accept Save_Hardware_Buffer_Character do

              Internal_Input_Buffer (
                Input_Buffer_Pointer ) :=
                  Hardware_Character_Buffer;

            end Save_Hardware_Buffer_Character;

            Input_Buffer_Pointer :=
              Input_Buffer_Pointer mod
                Max_Size_of_Internal_Input_Buffer + 1;

            Buffer_Count := Buffer_Count + 1;
      end select;
    end loop;
end Input_Penetrator_Task;
```

```
task body Output_Penetrator_Task is

    Max_Size_of_Internal_Output_Buffer : constant POSITIVE
      := 64;

    Internal_Output_Buffer :
      array ( 1 .. Max_Size_of_Internal_Output_Buffer )
        of CHARACTER;

    Input_Buffer_Pointer  : POSITIVE := 1;
    Output_Buffer_Pointer : POSITIVE := 1;

    Buffer_Count : INTEGER := 1;

    Hardware_Character_Buffer : CHARACTER;
    for Hardware_Character_Buffer use at 16#0200#;

    Hardware_Character_Buffer_is_Empty : BOOLEAN := TRUE;

  begin  -- Output_Penetrator_Task
    loop
      select
          accept Deposit_Character_into_Hardware_Buffer;

          if Buffer_Count > 0 then

              Hardware_Character_Buffer :=
                Internal_Output_Buffer (
                  Output_Buffer_Pointer );

              Output_Buffer_Pointer :=
                Output_Buffer_Pointer mod
                  Max_Size_of_Internal_Output_Buffer + 1;

              Buffer_Count := Buffer_Count - 1;

            else

              Hardware_Character_Buffer_is_Empty := TRUE;
          end if;

        or
          when Buffer_Count <
                Max_Size_of_Internal_Output_Buffer =>
              accept
                Put_Character_into_Penetrator_Output_Address
                    ( Char : in CHARACTER ) do

                  Internal_Output_Buffer (
                    Input_Buffer_Pointer ) := Char;

              end Put_Character_into_Penetrator_Output_Address;

              Input_Buffer_Pointer :=
                Input_Buffer_Pointer mod
                  Max_Size_of_Internal_Output_Buffer + 1;

              Buffer_Count := Buffer_Count + 1;

              if Hardware_Character_Buffer_is_Empty then

                Hardware_Character_Buffer :=
                  Internal_Output_Buffer (
                    Output_Buffer_Pointer );

                Output_Buffer_Pointer :=
                  Output_Buffer_Pointer mod
                    Max_Size_of_Internal_Output_Buffer + 1;

                Buffer_Count := Buffer_Count - 1;

                Hardware_Character_Buffer_is_Empty := TRUE;
              end if;
        end select;
      end loop;
  end Output_Penetrator_Task;


begin  -- Penetrate_Memory_Space

  -- . . .

  Input_Penetrator_Task.
    Get_Character_from_Penetrator_Input_Address ( Char );

  -- . . .

  Output_Penetrator_Task.
    Put_Character_into_Penetrator_Output_Address ( Char );
  -- . . .

end Penetrate_Memory_Space;
```

2.2.2.11 **System timing from package CALENDAR:**
Ada provides access to system timing through the package CALENDAR. If the accuracy of the system timing available through the package CALENDAR is inadequate for the needs of the TCB, then its security may be compromised (e.g., if the TCB has to run under real-time timing constraints). That is, the system timing available from package CALENDAR may not correspond precisely to the system clock. It should be noted that only system timing is available from package CALENDAR, which would be required for time stamping. To get timing from an external clock a new package must be developed, probably with representation clauses, which introduces their own complications, as discussed above. The potential dangers of this relationship between system timing and package CALENDAR should be the subject of future research with Ada runtime environments.

2.2.2.12 **Tailoring and Configuring Ada Compiler and Runtime System:**
Tailoring of an Ada compiler or runtime system is the actual modification of the code of the Ada compilation system [Baker 1988] at which time validation of the Ada compiler may be compromised and Trojan horses may be inserted. Configuration of an Ada compiler or runtime system is the reselection of compiler options and parameters provided by the Ada compiler vendor. Configuration may allow a compiler or runtime system to run conveniently on various host and target combinations. Unsafe selection of options and parameters may open paths for the TCB to be penetrated. Thus, modifying the Ada compiler or runtime system of a TCB by tailoring or configuring may compromise the security of the TCB.

2.3 Benefits of Using Tools to Develop Ada Software for TCB Systems

The following information on tools is taken liberally from Technology Training Corporation's seminar, Developing Ada Systems, presented by Mr. Jerry Mungle [Mungle 1988].

2.3.1 Desirable Features of Tools for the Development of Ada Software for TCB Systems

The following features of software development tools aid the development of Ada software for Ada TCB systems. No existing tool has all of these features. A highly desirable software development tool would meet the following requirements:

- Support the full software life cycle (including the special needs of real-time software development) from requirements through design, configuration management, and maintenance;
- Support various development methodologies, such as Object Oriented Development, PAMELA [Cherry 1984] and Structured Analysis Development;
- Support graphical design tools, e.g., Booch [1987A and 1987B] and Buhr [1984] Diagrams, PAMELA, Data Flow Diagrams, Control Flow Diagrams, and Structured Analysis;
- Support the generation of Ada Program (or Process) Design Language (PDL) and perhaps Diana Source Representation;
- Support the generation of data dictionaries, process specifications, and DOD-STD-2167A documentation;
- Support the generation of error reports;
- Have an Ada language-sensitive editor, a configurable Ada symbolic debugger, and a configurable Ada pretty printer;
- Support the tracing of the satisfaction of security requirements;
- Be available on various machines, e.g., VAX, SUN, APOLLO, IBM AT, and Macintosh.

2.3.2 Tools Available for the Development of Ada Software for TCB Systems

The following tools are available for the development of Ada systems. They do not inherently insure the development of a secure system.

A. AdaGRAPH:
   - It supports three methodologies: PAMELA, Object Oriented Development, and Structured Analysis Development.

- It provides Ada-specific support (one-to-one correspondence) of graphical design elements to Ada.
- It automatically generates Hierarchical Process Graphs (HPG), Software Architecture Data Bases, Ada PDL plus task idioms, Module Maps, and Data Dictionary.
- It runs on PC XT and AT and VAX/VMS.

B. AdaGEN:
- It supports Object Oriented Development.
- It provides Ada-specific support by drawing Booch diagrams, drawing Buhr diagrams, and generating compilable Ada PDL from diagrams.
- It runs on the PC and SUN computer.

C. TEAMWORK/Ada:
- It supports Ada Structure Graphs (Buhr graphs) and Structured Analysis Development.
- It automatically generates Ada code and DOD-STD-2167A documentation.
- It produces data flow diagrams, data dictionaries, process specifications, control flow diagrams, state transition diagrams, and state event matrices, decision tables, process activation tables, structure charts, and entity relationship diagrams.
- It interfaces with documentation production tools.
- It provides multiuser support on Apollo, DEC, Hewlett Packard, IBM, and SUN.

D. BYRON PDL:
- It supports unlimited user-definable report production.
- It produces Ada PDL, Diana source representation, and DOD-STD-2167 documentation.
- It provides program configuration management.

E. AISLE and ADADL:
- It is a family of tools that provides project management support, especially in the form of an extremely large number of reports. These reports include type cross references, error reports, complexity reports, and structure charts.
- It produces Ada PDL and code, DOD-STD-2167A documents, and data dictionaries.
- It provides an Ada code printer.
- It does not extend Ada.

F. DCDS:

- It supports the Distributed Computing Design System methodology, and it supports object oriented development and the spiral model of software development.
- It specifically supports large, complex, real-time, distributed systems.
- It supports the software life cycle from system requirements through integration and testing.
- Its process construction system supports building Ada code and automatically generates Ada data declarations.
- It supports software configuration management.
- It produces functional networks, requirements networks, Element-Relationship-Attribute (ERA) Specifications, data flow diagrams, N2 charts, and Ada PDL/code.
- It automatically generates DOD-STD-2167A documents.
- It runs on the VAX and SUN.

3.0 MAPPING OF ADA USAGE TO TCB CRITERIA

This section provides a mapping of Ada constructs that would be appropriate to the implementation of the TCB structures and functions defined in the TCSEC. The class B3 criteria are used as the template of the generalized TCB criteria because they are the highest level of security criteria under consideration. Also, the Ada constructs mapped to these criteria can similarly be mapped to the TCB criteria of lower security classes. The four TCB criteria subsections considered are Security Policy, Accountability, Assurance, and Documentation. For further discussion of the various topics, refer to Appendix A. All quotes are taken from the TCSEC.

GENERALIZED TCB CRITERIA

"The class B3 TCB must satisfy the reference monitor requirements that it mediate all accesses of objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. A security administrator is supported, audit mechanisms are expanded to signal security-relevant events, and system recovery procedures are required. The system is highly resistant to penetration."

3.1 Security Policy

A security policy describes how users may access documents or other information. It is the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

3.1.1 Discretionary Access Control

Discretionary access control provides a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) to any other subject (unless restrained by mandatory access control). An enforcement mechanism (e.g., access control lists) must allow users to specify and control sharing of those objects and must provide controls to limit propagation of access rights.

Application of Ada Constructs and Features:

Enforcement mechanisms that consist of lists, such as access control lists, can be created with linked lists and queues, using either static or dynamic storage constructs (e.g., using arrays or access types, respectively). Though linked lists and queues are more typically implemented using dynamic storage constructs created with Ada's access types, which provide more efficient memory usage, linked lists and queues created with arrays provide more efficient execution. In addition, the lists can be represented by abstract data types, which consist of Ada's strong data typing encapsulated in packages. For further explanation of these constructs, refer to Section 2.0.

Tasking would be necessitated if concurrent processes are to be used in more complex TCB systems where multiuser and multisubject requirements (e.g., simultaneously monitoring accesses to control lists by multiple subjects) are present. For further explanation of these constructs, refer back to Section 2.0.

Nonvolatile versions of major lists, e.g., access lists, need to be accessed from disk or tape, which require input and output operations whose security is protected with protocols that satisfy the class of the given TCB. For further explanation of these features, refer to Section 2.0.

3.1.2 Object Reuse

Object reuse is the reassignment to some subject of a medium (e.g., page frame disk sector, magnetic tape) that contained one or more objects. To be securely reassigned, such media must contain no residual data from the previously contained object(s). The TCB must assure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized.

Application of Ada Constructs and Features:

The reuse of objects involves the management of memory used for storing objects. This may involve the management of dynamic storage as well as static storage in a secure manner. Also, objects can be represented by abstract data types, which are implemented with Ada's packages and user-defined data types. Tasking and shared variables would be required to manage object reuse when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs, refer to Section 2.0.

3.1.3 Labels

A sensitivity label is a piece of information that represents the security level of an object and that describes the sensitivity (i.e., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions. They are associated with each ADP system resource (e.g., subject, storage object) that is directly or indirectly accessible by subjects external to the TCB, and they must be maintained by the TCB. To import non-labeled data, the TCB must request and receive from an authorized user the security level of the data, and all such actions must be auditable by the TCB. Also, the TCB must enforce subject sensitivity labels and device labels.

Application of Ada Constructs and Features:

Sensitivity labels can be treated as objects, which can be represented by abstract data types. These consist of Ada's strong data typing encapsulated in packages. The exportation of labeled information typically involves input and output using secure protocols, which can also be represented by abstract data types. Tasking and shared variables would be required to manage subject sensitivity labels and their exportation when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs, refer to Section 2.0.

3.1.4 Mandatory Access Control

A mandatory access control is a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. It prevents "some types of Trojan horse attacks by imposing access restrictions that cannot be bypassed, even indirectly. Under mandatory controls, the system assigns both subjects and objects special attributes that cannot be changed on request as can discretionary access control attributes such as access control lists. The system decides whether a subject can access an object by comparing their security attributes." [Gasser 1988, p.61] Thus, a TCB must enforce a mandatory access control policy over all resources (i.e., subjects, storage objects and I/O devices) that are directly or indirectly accessible by subjects external to the TCB. All subjects and storage objects must be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels must be the basis of the mandatory access control decisions. Also, a TCB must be able to support two or more security levels.

Application of Ada Constructs and Features:

The management of a mandatory access control policy is performed typically with an implementation of the Bell and LaPadula security model that regulates the security of accessing objects by subjects and the assignment of sensitivity labels to enforce the policy. This requires classifications of the objects that are to be protected by this policy. This may involve the management of dynamic and static storage in a secure manner. Also, objects can be represented by abstract data types, which are implemented with Ada's packages and user-defined data types. Similarly, the policy could be represented by a package. Tasking and shared variables would be required to manage mandatory access controls when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs, refer to Section 2.0.

3.2 Accountability

Accountability is the monitoring of access to and operation of a TCB system by using identification and authentication of users requesting access to the system, maintenance of trusted communication paths, and auditing of accesses to the TCB.

3.2.1 Identification and Authentication

Identification consists of using unique identifiers that are associated with each user (such as a last name, initials, or account number), that everyone knows, that no one can forge or change, and that all access requests can be checked against. The identifier is the means by which the system distinguishes users. In particular, a TCB must require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate.

Authentication consists of associating a real user (or more accurately, a program running on behalf of a user) with a unique identifier, namely, the user ID. "The system must separate authentication information (passwords) from identification information (unique IDs) to the maximum extent possible because passwords are secret and user IDs are public" [Gasser 1988, p.23]. A TCB must maintain authentication data that includes information for verifying the identity of individual users as well as information for determining the clearance and authorizations of individual users. It uses this data to authenticate the user's identity and to determine the security level and authorizations of subjects that are created to act on behalf of the individual user. The TCB must protect authentication data so that it cannot be accessed by an unauthorized user. It must be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. Also, it must provide the capability of associating the individual identity with all auditable actions taken by that individual.

Application of Ada Constructs and Features:

The management of identification and authentication data involves using corresponding abstract data types, which consist of Ada's strong data typing encapsulated in packages. This management may also include using dynamic storage as well as static storage in a secure manner. Similarly, the identification and authentication processes may be represented by packages. Tasking and shared variables may be required to manage security data when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs, refer to Section 2.0.

3.2.2 Trusted Path

A trusted path is a mechanism by which a person at a terminal can communicate directly with the TCB. This mechanism can only be activated by the person or the TCB and cannot be imitated by unevaluated software. A TCB must support a trusted communication path between itself and users for use when a positive TCB-to-user connection is required (e.g., login, change subject security level). Communication via this trusted path must be activated exclusively by a user or the TCB and must be logically isolated and unmistakably distinguishable from other paths. The Trusted Path for a Class B3 TCB needs to allow bi-directional access, from user to TCB and from TCB to user.

Application of Ada Constructs and Features:

Trusted paths require secure input and output communication paths between the user or subject and the TCB. Trusted paths can be treated as objects, which can be represented by abstract data types. These consist of Ada's strong data typing encapsulated in packages. Tasking and shared variables may be required to manage trusted paths when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs and features, refer to Section 2.0.

3.2.3 Audit

An audit of accesses to and operation of TCB operation is maintained in a set of records (i.e., an audit trail) that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backward from records and reports to their component source transactions. The TCB must be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects. Audit data must be protected by the TCB so that read access to it is limited to those who are authorized for audit data. A TCB must be able to record use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events. A TCB must be able to audit any override of human-readable output markings. For each recorded event, the audit record must identify the date and time of event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record must include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion events, the audit record must include the name of the object and the object's security level. The ADP system administrator must be able to selectively audit the actions of any one or more users based on individual identity and/or object security level. A TCB must be able to audit the identified events that may be used in the exploitation of covert storage channels. A TCB must contain a mechanism that is able to monitor the occurrence or accumulation of security auditable events that may indicate an imminent violation of security policy; this mechanism must be able to immediately notify the security administrator when thresholds are exceeded. If the occurrence or accumulation of these security relevant events continues, the system must take the least disruptive action to terminate the event.

Application of Ada Constructs and Features:

Managing audit data requires maintaining an audit trail. An audit trail is typically recorded on disk and/or tape, which requires secure input and output communication paths within a TCB that includes a secure disk and/or tape. The audit data and audit trail can be treated as objects, which can be represented by abstract data types. These consist of Ada's strong data typing encapsulated in packages. Also, the audit data may be (at least partially) located in dynamic storage. Tasking and shared variables may be required to manage the audit data and audit trail when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. For further explanation of these constructs and features, refer to Section 2.0.

3.3 Assurance

Assurance of the correctness of a TCB system's security controls, as specified by the security requirements, determines the extent that the security architecture must dictate many details of the development process. The two types of assurance that must be considered are operational and life-cycle.

3.3.1 Operational Assurance

Operational assurance includes the following aspects: system architecture, system integrity, covert channel analysis, trusted facility management, and trusted recovery.

3.3.1.1 System Architecture

The TCB must maintain a domain for its own execution that protects it from external interference or tampering. It must maintain process isolation through the provision of distinct address spaces under its control. The TCB must be internally structured into well-defined largely independent modules. It must make effective use of available hardware to separate those elements that are protection-critical from those that are not. TCB modules must be designed such that the principle of least privilege is enforced. Features in hardware, such as segmentation, must be used to support logically distinct storage objects with separate attributes (namely, readable and writable). The TCB user interface must be completely defined and all elements of the TCB must be identified. The TCB must be designed and structured to use a complete, conceptually simple protection mechanism with precisely defined semantics; this mechanism must play a central role in enforcing the internal structuring of the TCB and the system. The TCB must incorporate significant use of layering, abstraction, and data hiding. Significant system engineering must be directed toward minimizing the complexity of the TCB and excluding from the TCB modules that are not protection-critical.

Application of Ada Constructs and Features:

The TCB's system architecture must be modular, and use data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types. To ensure system integrity and to prevent the creation of covert channels, the creation of TCB system features (dynamic storage, input and output communications within the TCB and between the user or subject and the TCB, tasking and/or global and shared variables) must address the potential security problems associated with their implementation and use. For further explanation of these constructs and features, refer to Section 2.0.

3.3.1.2 System Integrity

Hardware and/or software features must be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB.

Application of Ada Constructs and Features:

Ada allows for the cre·:tion of more readable code which helps the validation process.

3.3.1.3 Covert Channel Analysis

The search for covert channels required in this analysis is facilitated by having the TCB's software and documentation be readable and understandable.

Application of Ada Constructs and Features:

Ada allows for the creation of more readable code, which helps the identification of covert channels.

3.3.1.4 Trusted Facility Management

The majority of issues related to trusted facility management are not software issues. Rather, they are concerned with the responsibilities of the security administrator and the ADP system administrative personnel.

Application of Ada Constructs and Features:

Ada allows for the creation of more readable code, which helps the management of a trusted facility.

3.3.1.5 Trusted Recovery

Procedures and/or mechanisms must be provided to assure that, after an ADP system failure or other discontinuity, recovery without a protection compromise is obtained.

Application of Ada Constructs and Features:

When implemented with proper security considerations, Ada's exception handling mechanism can be used to help implement trusted recovery. For further explanation of these constructs, refer to Section 2.0.

3.3.2 Life-Cycle Assurance

Life-Cycle assurance includes the following aspects: security testing, design specification and verification, and configuration management.

3.3.2.1 Security Testing

Security mechanisms of the ADP system must be tested and found to work as claimed in the system documentation. A team of individuals who thoroughly understand the specific implementation of the TCB must subject its design documentation, source code, and object code to thorough analysis and testing. The team's objective should be to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy. This will assure that no subject is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users. The TCB must be found resistant to penetration. All discovered flaws must be corrected, and the TCB must be retested to demonstrate that the flaws have been eliminated and that new flaws have not been introduced. Testing must demonstrate that the TCB implementation is consistent with the descriptive top-level specification. No design flaws and no more than a few correctable implementation flaws may be found during testing and there must be reasonable confidence that few remain.

Application of Ada Constructs and Features:

Security testing of the TCB system is promoted by having the system architecture exhibit modularity and using data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types. Security testing of the TCB system must include the testing of all implementations of the following system features: dynamic storage management, input and output communications within the TCB and between the user or subject and the TCB, tasking and/or global and shared variables. For further explanation of these constructs and features, refer to Section 2.0.

3.3.2.2 Design Specification and Verification

A formal model of the security policy supported by the TCB must be maintained and be proven to be consistent with its axioms. A descriptive top-level specification (DTLS) of the TCB must be maintained that completely and accurately describes the TCB in terms of exceptions, error messages, and effects. It must be shown to be an accurate description of the TCB interface. A convincing argument must be given that the DTLS is consistent with the model.

3.3.2.3 Configuration Management

A configuration management system must be in place that maintains control of changes to the descriptive top-level specification, other design data, implementation documentation, source code, the running version of the object code, and test fixtures and documentation. The configuration management system must assure a consistent mapping among all documentation and code associated with the current version of the TCB. Tools must be provided for generation of a new version of the TCB from source code and for comparing a newly generated version with the previous TCB version in order to ascertain that only the intended changes have been made in the code that will actually be used as the new version of the TCB.

3.4 Documentation

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. While this is true, no special consideration needs to be given to the development of the documentation for a TCB. The documentation must be developed to meet all requirements set forth in the TCSEC and must be complete and up to date.

Application of Ada Constructs and Features:

The documentation, particularly the design documentation, must clearly convey the implementation of the TCB system architecture, which must exhibit modularity, and use data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types. These features not only aid the understandability of the code, but also of the design documentation. The documentation must clearly show how the security risks are handled, including those posed by the following TCB system features: dynamic storage management, input and output communications within the TCB and between the user or subject and the TCB, tasking and/or global and shared variables, and any tailoring or configuring of the Ada compiler or runtime system. This discussion of the application of Ada constructs to documentation applies to the following four subsections: Security Features User's Guide, Trusted Facility Manual, Test Documentation, and Design Documentation. For further explanation of these constructs and features, refer to Section 2.0.

Ada's self-documenting capabilities can contribute to the design and testing documentation. The self-documenting capabilities can be realized better with the use of readable and understandable mnemonics. The code should exhibit consistent indentation. Blank lines should be used to logically partition the code. Comments should be inserted into the code to provide information that is not conveyed by the code. Each package and subprogram should have a header that states its purpose, its authors, and the history (dates) of its creation and revision(s).

3.4.1 Security Features User's Guide

A single summary, chapter, or manual in user documentation must describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another.

Application of Ada Constructs and Features:

Ada will have no impact on the development of the Security Features User's Guide.

3.4.2 Trusted Facility Manual

A manual addressed to the ADP system administrator must present cautions about functions and privileges that should be controlled when running a secure facility. It must describe the operator and administrator functions related to security, to include changing the security characteristics of a user. The manual must provide guidelines on the consistent and effective use of the protection features of the system, how they interact, how to securely generate a new TCB, and facility procedures, warnings, and privileges that need to be controlled in order to operate the facility in a secure manner. TCB modules that contain the reference validation mechanism must be identified. Procedures for secure generation of a new TCB from source after modification of any modules in the TCB must be described. The manual must include the procedures to ensure that the system is initially started in a secure manner. Procedures must also be included to resume secure system operation after any lapse in system operation.

Application of Ada Constructs and Features:

Ada will have no impact on the development of the Trusted Facility Manual.

3.4.3 Test Documentation

The system developer must provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing. Documentation must include results of testing the effectiveness of the methods used to reduce covert channel bandwidths.

Application of Ada Constructs and Features:

Ada coding should be self-documenting, as discussed in Section 3.4, so that it can contribute to the testing documentation.

3.4.4 Design Documentation

Documentation must be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB. The interfaces between the TCB modules must be described. A formal description of the security policy model enforced by the TCB must be available and proven that is sufficient to enforce the security policy. Specific TCB protection mechanisms must be identified, and an explanation must be given to show that they satisfy the model. The TCB implementation (i.e., in hardware, firmware, and software) must be shown, using informal techniques, to be consistent with the DTLS. The elements of the DTLS must be shown, using informal techniques, to correspond to the elements of the TCB. Documentation must describe how the TCB implements the reference monitor concept and give an explanation why it is tamper resistant, cannot be bypassed, and is correctly implemented. Documentation must describe how the TCB is structured to facilitate testing and to enforce least privilege. Documentation must also present the results of covert channel analysis and the tradeoffs involved in restricting the channels. All auditable events that may be used in the exploitation of known covert storage channels must be identified. The bandwidths of known covert storage channels, the use of which is not detectable by the auditing mechanisms, must be provided.

Application of Ada Constructs and Features:

Ada coding should be self-documenting, as discussed in Section 3.4, so that it can contribute to the design documentation. The use of Ada PDL will assist in generating readable and understandable design documentation.

4.0 SUMMARY AND CONCLUSIONS

4.1 Summary

This appendix identified benefits and potential deterrents of using Ada in the software development of TCBs. These benefits included Ada's assets in the application of sound software engineering principles, such as data abstraction, information hiding, modularity, and localization. Also included among the benefits were such Ada constructs as strong data typing, packages, subprograms, and tasks. This appendix also identified and categorized potential deterrents of using Ada in the software development of TCBs. These included shortcomings inherent to programming languages in general, shortcomings unique to the Ada language, and benefits of using tools to develop Ada software.

Sections 2.0 and 3.0 addressed the issues in two ways. First, Section 2.0 focused on the following issues: (1) Benefits of Using Ada in Developing TCB Systems, (2) Potential Deterrents to Using Ada in Developing TCB systems, (3) Shortcomings inherent to programming languages in general in developing TCB systems, (4) Shortcomings unique to the Ada language in developing TCB systems, (5) Benefits of using tools for developing Ada software for TCB systems. Section 3.0 presented these issues in the context of a mapping of Ada usage to generalized TCB criteria. Ada constructs were identified that may be used to implement TCB features and functions.

4.2 Conclusions

Because Ada was designed with features and constructs that promote recognized sound software engineering principles, more so than other current HOLs, it is well suited as the implementation language of TCBs. Although Ada offers various specific benefits, the potential deterrents of using Ada must be recognized and addressed. Though these potential problems were identified with using various Ada features, this is not meant to imply that any of the features should not be used. Rather, the security of the TCB must not be compromised, as discussed in Section 2.0, when any of the constructs and features are used in the implementation of the TCB.

5.0 BIBLIOGRAPHY

Abrams, Marshall D., Podell, Harold J., 1987. Tutorial Computer and Network Security. Washington, D. C.: IEEE Computer Science Press.

Abrams, Marshall D., Podell, Harold J., 1988. Recent Developments in Network Security. 2906 Covington Road, Silver Spring, MD, 20910.: Computer Educators Inc.

Anderson, Eric R., Ada's Suitability for Trusted Computer Systems from Proceedings of the Symposium on Security and Privacy, Oakland, California, 22-24 April, 1985.

Baker, T. P., 13 July 1988. Issues Involved in Developing Real-Time Ada Systems. Department of Computer Science, Florida State University, Tallahasse, FL: for U. S. Army HQ, COMM/ADP.

Boebert, W. E., Kain, R. Y., and Young, W. D., July 1985. "Secure Computing: The Secure Ada Target Approach." Scientific Honeyweller, Vol. 6, No. 2.

Boehm, Barry. 1988. A Spiral Model of Software Development and Enhancement. Washington, DC: IEEE Computer Science Press.

Booch, Grady. 1987A. Software Components with Ada. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Booch, Grady. 1987B. Software Engineering with Ada. 2nd ed. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Brill, Alan E., 1983. Building Controls Into Structured Systems. New York, N. Y.: YOURDON Press Inc.

Buhr, R. J. A., 1984. System Design with Ada. Englewood Cliffs, N. J.: Prentice-Hall.

Cherry, George W., 1984. Parallel Programming in ANSI Standard Ada. Reston, Virginia: Reston Publishing Company, Inc.

Defense System Software Development. 1988. DoD-STD-2167A, 29 February 1988.

Feldman, Michael B., 1985. Data Structures with Ada. Reston, Virginia: Reston Publishing Company, Inc.

Freeman, Peter. 1987. Tutorial: Software Reusability. Washington, D. C.: IEEE Computer Science Press.

Final Evaluation Report of SCOMP 23 September 1985. Secure Communications Processor STOP Release 2.1.

Gasser, Morrie. 1988. Building a Secure Computer System. New York, N. Y.: Van Nostrand Reinhold Company, Inc.

Gehani, Narain. 1984. Ada Concurrent Programming. Englewood Cliffs, N. J.: Prentice-Hall Inc.

Gilpin, Geoff. 1986. Ada: A Guided Tour and Tutorial. New York, N. Y.: Prentice Hall Press.

IEEE Standard Glossary of Software Engineering Terminology. 18 February 1983. (IEEE Std 729-1983).

Mungle, Jerry. 1988. Developing Ada Systems. Technology Training Corporation's seminar.

National Computer Security Center. 1985. Department of Defense Trusted Computer System Evaluation Criteria. (DOD 5200.28-STD)

National Computer Security Center. 1987. Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria.

Nissen, John and Wallis, Peter. 1984. Portability and Style in Ada. Cambridge, Great Britain: Cambridge University Press.

Reference Manual for the Ada Programming Language. 1983. ANSI/MIL-STD-1815A-1983, 17 February 1983.

Ross, D. T., Goodenough, J. B., and Irvine, C. A., 1975. "Software Engineering: Process, Principles, and Goals," Computer.

Saydjari, O. S., Beckman, J. M., and Leaman, J. R., 1987. "LOCKing Computers Securely," Proceedings, 10th National Computer Security Conference, Baltimore, MD: September 21-24, 1987, National Bureau of Standards/National Computer Security Center.

Shaffer, Mark of Honeywell, Computing Technology Center, and Walsh, Geoff of R & D Associates Secure. 1988. "LOCK/ix: On Implementing Unix on the LOCK TCB," Proceedings, 11th National Computer Security Conference, Baltimore, MD: October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.

Tracz, Will. 1988. Tutorial: Software Reuse: Emerging Technology. Washington, D. C.: IEEE Computer Science Press.

Trusted Computer System Security Requirements Guide for DoD Applications. ?
September 1987.

APPENDIX C

Programming Guidelines for Using Ada in the Software Development of Trusted Computing Bases

Prepared for:

National Computer Security Center
9800 Savage Road
Fort Meade, MD 20755

Prepared by:

Ada Applications And
Software Technology Group

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

April 1989

APPENDIX C
TABLE OF CONTENTS

1.0 INTRODUCTION

This appendix presents guidelines for developing trusted computing bases (TCBs) in Ada. It is meant to complement existing standards. The user of this appendix should also use standards or guidelines for the development of TCBs. General TCB development issues are discussed in this document only to the extent that they are affected by Ada. Also the developer of a TCB in Ada should use general purpose Ada guidelines for direction on the general use of Ada features. These guidelines detail the use of Ada features only as they would be used for specific aspects of TCB development. This document, therefore, is to complement existing TCB development guidelines and general-purpose Ada coding guidelines.

1.1 Background

This appendix provides guidelines on how to use Ada in the development of TCB systems. This guidance focuses on how to exploit the advantages of using Ada, such as data abstraction, information hiding, modularity, localization, strong data typing, packages, subprograms, and tasks.

This appendix does not imply that software alone is sufficient for ensuring security in a TCB. As discussed in the papers "Secure Computing: The Secure Ada Target Approach," "LOCKing Computers Securely," and "LOCK/ix: On Implementing Unix on the LOCK TCB," the security of a system cannot be insured with software alone. Hardware is fundamentally important to TCB system security. This appendix does not discuss hardware aspects of TCB systems. If the reader wishes to investigate the hardware issues, he should consult the references cited above.

1.2 Format

This appendix consists of four sections. The first section is an introduction which consists of background information and definitions of key terms that appear in this document. The key terms section includes terms found in the glossary of the TCSEC, the Reference Manual for the Ada Programming Language (LRM) [ANSI/MIL-STD-1815A-1983], and the IEEE Standard Glossary of Software Engineering Terminology [IEEE 1983]. Sections 2.0 and 3.0 address the guidelines in two ways. First, Section 2.0 provides definitions of and general programming guidelines on the use of Ada constructs and features that have significant bearing on the development of TCB Systems. These guidelines are mapped to the LRM. Several Ada constructs are virtually or literally identical to those of other high order languages such as FORTRAN and C. In these instances, the phrase "No Ada-specific impact on TCBs" appears in the text. Also, some subsection topics are addressed by the main section under which they fall. When the subsection adds no additional effect beyond what is discussed in the main section the phrase "No additional Ada-specific impact on TCBs" appears. Section 3.0 provides guidelines on the application of Ada constructs and features in the context of a mapping of Ada usage to generalized TCB criteria. In particular, class B3 as defined in the TCSEC is used as a template for the generalized TCB criteria. Though potential problems exist when using various Ada constructs and features, this does not mean that any of the constructs and features should not be used. Rather, certain sets of features when used together must be used in accordance with the established guidelines to ensure the security of the TCB. Section 4.0 consists of a summary of this appendix and its conclusions.

1.3 Key terms

Several key words appear throughout the text of this document. These words have specific meanings within the context of certified systems and their definitions are presented here.

The following definitions are taken directly from the TCSEC:

Access - A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

Audit Trail - A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions.

Covert Channel - A communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Data - Information with a specific physical representation.

Discretionary Access Control - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Mandatory Access Control - A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

Object - A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc.

Sensitivity Label - A piece of information that represents the security level of an object and that describes the sensitivity (e.g., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions.

Subject - An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair.

Trusted Computing Base (TCB) - The totality of protection mechanisms within a computer system -- including hardware firmware, and software -- the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Additional Terms

These terms are included because they appear frequently in the following text.

Pragma - A compiler directive. That is, it "is used to convey information to the compiler." According to the Reference Manual for the Ada Programming Language (LRM) [ANSI/MIL-STD-1815A-1983], the predefined pragmas (Refer to Annex B in this manual for descriptions) "must be supported by every implementation. In addition, an implementation may provide implementation-defined pragmas, which must then be described in Appendix F," i.e., the appendix on implementation-dependent characteristics that the Ada compiler vendor must provide in his Ada language reference manual.

Security - The protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure. [IEEE 1983] Security also pertains to personnel, data, communications, and the physical protection of computer installations. Specifically, for the purposes of this report, security is defined by the criteria in the TCSEC, i.e., a given security problem corresponds with a specific TCSEC criteria.

2.0 MAPPING OF TCB RELEVANT ADA CONSTRUCTS AND FEATURES TO THE REFERENCE MANUAL FOR THE ADA PROGRAMMING LANGUAGE

This section provides software engineering and programming guidelines for using Ada constructs and features in developing TCB systems. These guidelines are mapped to the LRM [ANSI/MIL-STD-1815A-1983] to provide a convenient means of reference. These guidelines are for software engineering practices and programming conventions for using the Ada language to develop TCB systems. Although these guidelines are recommendations, justification should be made for any deviation from them. The development of any high quality system, and especially that of a TCB system, requires strict adherence to sound software engineering principles and practices, from requirements analysis and design through coding and maintenance.

Ada supports sound software engineering principles, including the following (Discussions on them are located in the indicated sections.):

Strong Data Typing	(3. Declarations and Types)
Data Abstraction	(7. Packages)
Information Hiding	(7. Packages)
Modularity and Localization	(7. Packages)
Reusable Code	(10.4 The Program Library)

Note, a discussion on tailoring and configuring Ada compiler and runtime systems is located in Section 2.1.

Although this appendix does not advocate abstaining from using any Ada constructs, it should be noted that each of them, to varying degrees, contributes to a large Ada runtime library. Eric Anderson, in his paper "Ada's Suitability for Trusted Computer Systems," proposes the extreme view of minimizing the Ada runtime library for the security kernel as a primary goal because its size may lend itself to hiding covert channels and Trojan horses. To minimize the size of the Ada runtime library, he proposes the following severe restrictions in creating the security kernel: not allowing use of any dynamic storage, tasking, exception handling, any Ada standard packages other than STANDARD and SYSTEM; limiting the use of runtime constraint checking, math and conversion routines, and representation clauses. Addressing security issues associated with the Ada runtime library are beyond the scope of this document. Clearly, though, Ada runtime library security issues need to be addressed by further research.

Guideline:

1. Coding guidelines identified in the remainder of Section 2.0 should be used in conjunction with general software engineering standards and with Ada coding standards. The guidelines presented here address only the use of Ada for specific TCB criteria.

All but one of the remaining subsections of this appendix are structured in accordance with the chapters of the LRM. The remaining subsection covers tailoring and configuring Ada compilers and runtime systems. Each subsection includes descriptions of an Ada construct or feature, programming guidelines for using it in developing TCB systems, and examples as needed.

More general coding standard issues, such as limiting the amount of nesting of loop and if blocks and always providing an others clause in case statements, are not addressed by the guidelines presented here.

2 - 1. Introduction

This chapter of the LRM introduces the Ada language standard. The two sections of primary interest to developing TCBs are Scope of the Standard (Section 2 - 1.1), which discusses Ada standardization and portability, and Classification of Errors (Section 2 - 1.6).

2 - 1.1 Scope of the Standard

This definition of the Ada programming language was developed to promote Ada's standardization and portability. Because Ada, more so than other languages, is very standardized (in particular by the Department of Defense (DoD)) no subsets or supersets of Ada are allowed. This standardization is assured by the DoD's process of validation of Ada compilers. This aids the portability of Ada software among different types of computers. In particular, an Ada TCB system is likely to be easier to port between different types of computers than if the TCB were developed in another language.

Guidelines:

1. To take advantage of Ada's standardization and portability the use of the following items should be avoided: representation clauses, implementation-dependent features, and interfacing with other languages.
2. Issues involving such features, which may compromise standardized code and/or code portability, should be addressed as early as possible in the development of a TCB system, preferably in the TCB's requirements specification.
3. The implementation of these features should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 1.1.1 Extent of the Standard

No additional Ada-specific impact on TCBs

2 - 1.1.2 Conformity of an Implementation with the Standard

No additional Ada-specific impact on TCBs

2 - 1.2 Structure of the Standard

No Ada-specific impact on TCBs

2 - 1.3 Design Goals and Sources

No Ada-specific impact on TCBs

2 - 1.4 Language Summary

No Ada-specific impact on TCBs

2 - 1.5 Method of Description and Syntax Notion

No Ada-specific impact on TCBs

2 - 1.6 Classification of Errors

The LRM defines four types of errors: compilation time errors, runtime errors, erroneous execution, and incorrect order dependencies. The first of these four types of errors, compilation time errors, are commonly referred to as syntax errors, and will not allow for compilation of the program. The second type of error, the runtime error, occurs during the attempted execution of the program, and is commonly referred to as an "exception." Runtime errors in general have been widely discussed in the literature [Goodenough and Liskov 1985], and have even been discussed with respect to Ada [Luckham 1980]. In addition to this, because Chapter 7, "Exceptions," is devoted entirely to this topic, its discussion will be deferred to that chapter. The remaining error types, erroneous execution and incorrect order dependencies, are not required to be detected at compilation or execution, but do result in violations of certain rules of the Ada language.

Guidelines:

1. The software developer, rather than the Ada language, is responsible for the detection of the four types of errors: compilation time errors, runtime errors, erroneous execution, and incorrect order dependencies.

2. The detection of these four types of errors should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 2. Lexical Elements

This chapter of the LRM defines the lexical elements allowed in the text of an Ada compilation unit. It also describes pragmas, which provide certain information for the compiler. The majority of these items have little relevance to the software development of TCBs.

2 - 2.1 Character Set

No Ada-specific impact on TCBs

2 - 2.2 Lexical Elements, Separators, and Delimiters

No Ada-specific impact on TCBs

2 - 2.3 Identifiers

An important key to the readability of software is the use of mnemonic identifiers. Readability of software is a key component of software understandability.

Guidelines:

1. Name identifiers with clearly readable and understandable mnemonics.
2. Tailor the naming of identifiers to the application to improve the readability of programs and reduce the chance for errors [ASOS 1987].

Examples: The following examples illustrate clearly readable and understandable mnemonic identifier names. They are identifiers that are used in later examples in this appendix. The use of each identifier should be clear based on its name. This set of identifiers is taken out of context; therefore, it does not constitute compilable code.

```
Named_Object_MAC_Record         -- see 2 - 3.2.1
Max_Named_String_Length         -- see 2 - 3.2.2
Named_Individuals_List_Type     -- see 2 - 3.2.1
Scrubbed_Named_Objects_List     -- see 2 - 3.2.1
```

2 - 2.4 Numeric Literals

Guideline:
1. Use underscore in numeric literals to add clarity and thus reduce the number of accidental errors [ASOS 1987].

Examples: This example illustrates underscoring in numeric literals which improve the readability of literals that have many digits. Underscores are used in place of commas in large numbers. In strings of digits to the right of the decimal, an underscore is used, for example, after every fifth digit.

```
123_456 rather than 123456          -- integer literal
3.14159_26 rather than 3.1415926    -- real literal
```

2 - 2.4.1 Decimal Literals

No Ada-specific impact on TCBs

2 - 2.4.2 Based Literals

Guideline:
1. Use base literals to allow clear expression of bit masks and other items not easily expressed in decimal notation [ASOS 1987].

Examples: This example illustrates base literals.

```
-- integer literals of value 255
2#1111_1111#   16#FF#    016#0FF#
```

2 - 2.5 Character Literals

No Ada-specific impact on TCBs

2 - 2.6 String Literals

No Ada-specific impact on TCBs

2 - 2.7 Comments

Comments should provide additional instructive information about modules and their underlying algorithms that is not conveyed by their code.

Guideline:
1. The information provided in comments should reflect the TCB's design, and promote the TCB's coding and maintenance.

2 - 2.8 Pragmas

The **PRAGMA** construct is relevant, although this chapter of the LRM does not discuss its use and application, but only rules for its placement within the program text. For the discussion of its use and application see Section 2-11.7.

2 - 2.9 Reserved Words

No Ada-specific impact on TCBs

2 - 2.10 Allowable Replacement of Characters

No Ada-specific impact on TCBs

2 - 3. Declarations and Types

This chapter of the LRM defines the type mechanism, the means for declaring objects of the types, and the set of operations on the types. The major areas that are of concern to the development of TCBs are object and type declarations, array types, and access types. Static storage is discussed in **Array Types** (Section 2 - 3.6). Dynamic storage is discussed in **Access Types** (Section 2 - 3.8).

Ada's strong data typing facilities provide a means of associating related data objects with data types, which is a fundamental aspect of data abstraction. Strong data typing serves to isolate data types. For a further discussion of data abstraction refer to Section 2 - 7, **Packages**.

Guidelines:
1. Use Ada's strong data typing to create user-defined types, namely, subtypes and derived types.

2. Name data types and objects with clearly readable and understandable mnemonics that promote the maintenance of the code.

2 - 3.1 Declarations

No Ada-specific impact on TCBs

2 - 3.2 Objects and Named Numbers

2 - 3.2.1 Object Declarations

An erroneous program may occur if it includes the use of an object prior to assigning a value to the object. The formal definition of the language requires that the default value for objects be specified when declared.

Guideline:
1. Initialize data items in their declarations [ASOS 1987].

2. Disallow references to objects before their initialization. This should be addressed with explicit initialization and monitored during code walkthroughs. That is, ensure that no undefined variable will be referenced by assigning a default value in each object declaration.

3. Name all constants and literals [ASOS 1987].

Examples: The following examples illustrate clearly readable and understandable mnemonic object declarations. This set of object declarations includes the initialization of the objects.

```
--      Max_Named_String_Length, Name_String_Type, and Blank_Name_String
--      are declared in package Basic_TCB_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included formally in
--      this appendix.

        Blank_Name_String : Name_String_Type                    -- see 2 - 3.6.3
          := ( 1 .. Max_Named_String_Length => ' ');            -- see 2 - 3.2.2


--      The next three declarations in this section are located in the array
--      types version of package Access_Control_List_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        Scrubbed_Named_Objects_List :
                 Named_Objects_List_Type                        -- see 2 - 3.6
             := Named_Objects_List_Type'
                  ( others => Basic_TCB_Types_Package.
                                 Blank_Name_String );

        Scrubbed_Named_Individuals_List :
                 Named_Individuals_List_Type                    -- see 2 - 3.6
             := Named_Individuals_List_Type'
                  ( others => Basic_TCB_Types_Package.
                                 Blank_Name_String );

        Scrubbed_Groups_of_Named_Individuals_List :
                 Groups_of_Named_Individuals_List_Type          -- see 2 - 3.6
             := Groups_of_Named_Individuals_List_Type'
                  ( others => Basic_TCB_Types_Package.
                                 Blank_Name_String );


--      The next three declarations in this section are located in the access
--      types version of package Access_Control_List_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.
```

```
Scrubbed_Named_Objects_List :
        Named_Objects_List_Type                         -- see 2 - 3.8
     := Named_Objects_List_Type'
         ( Name => Basic_TCB_Types_Package.
                     Blank_Name_String,
           Next => null );

Scrubbed_Named_Individuals_List :
        Named_Individuals_List_Type                     -- see 2 - 3.8
     := Named_Individuals_List_Type'
         ( Name => Basic_TCB_Types_Package.
                     Blank_Name_String,
           Next => null );

Scrubbed_Groups_of_Named_Individuals_List :
        Groups_of_Named_Individuals_List_Type           -- see 2 - 3.8
     := Groups_of_Named_Individuals_List_Type'
         ( Name => Basic_TCB_Types_Package.
                     Blank_Name_String,
           Next => null );


--      The remaining declarations in this section are located
--      in both the access and array types versions of
--      package Access_Control_List_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included formally in
--      this appendix.

Scrubbed_Named_Object_ACL_Record :
  Named_Object_Record_Type                              -- see 2 - 3.7
    := Named_Object_Record_Type'
        ( Name => Basic_TCB_Types_Package.
                    Blank_Name_String,

          Sensitivity_Label =>
            Mandatory_Access_Control_Types_Package.
              Scrubbed_Sensitivity_Label,                -- see 3 - 3.1.3

          Authorized_Named_Individuals_List =>
            Scrubbed_Named_Individuals_List,

          Authorized_Groups_of_Named_Individuals_List =>
            Scrubbed_Groups_of_Named_Individuals_List,

          Unauthorized_Named_Individuals_List =>
              Scrubbed_Named_Individuals_List );
```

```
Named_Object_ACL_Record : Named_Object_Record_Type  -- see 2 - 3.7
    := Scrubbed_Named_Object_ACL_Record;

Scrubbed_Named_Individual_ACL_Record :
  Named_Individual_Record_Type                       -- see 2 - 3.7
    := Named_Individual_Record_Type'
         ( Name              => Basic_TCB_Types_Package.
                                  Blank_Name_String,
                                                      -- see 2 - 3.6.3
           Sensitivity_Label =>
             Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label,             -- see 3 - 3.1.3

           Named_Objects_List => Scrubbed_Named_Objects_List );


Named_Individual_ACL_Record :
  Named_Individual_Record_Type                       -- see 2 - 3.7
    := Scrubbed_Named_Individual_ACL_Record;



Scrubbed_Group_of_Named_Individuals_ACL_Record :
  Group_of_Named_Individuals_Record_Type             -- see 2 - 3.7
    := Group_of_Named_Individuals_Record_Type'
         ( Name              => Basic_TCB_Types_Package.
                                  Blank_Name_String,
                                                      -- see 2 - 3.6.3
           Sensitivity_Label =>
             Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label,            -- see 3 - 3.1.3

           Named_Objects_List => Scrubbed_Named_Objects_List );


Group_of_Named_Individuals_ACL_Record :
  Group_of_Named_Individuals_Record_Type             -- see 2 - 3.7
    := Scrubbed_Group_of_Named_Individuals_ACL_Record;
```

2 - 3.2.2 Number Declarations

The use of number declarations improve the readability, understandability, and maintenance of code.

Guideline:
1. Use mnemonic number declarations to make code more readable and understandable.

Example: The following example illustrates a clearly readable and understandable mnemonic number declaration

```
--      Max_Named_String_Length is declared in package Basic_TCB_Types_Package.
--      Because only essential features of this package need to be shown, it is
--      not included formally in this appendix.

        Max_Named_String_Length : constant := 80;
```

2 - 3.3 Types and Subtypes

Although types and subtypes within Ada are well behaved with respect to developing TCBs. Memory management of objects of array types and access types warrants special consideration.

2 - 3.3.1 Type Declarations

Example: The following example illustrates a clearly readable and understandable mnemonic type declaration.

```
--      Day_Type is declared in package Basic_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        type Day_Type is ( Sunday, Monday, Tuesday, Wednesday,
                           Thursday, Friday, Saturday );
```

For additional discussion refer to Section 2 - 3. and 2 - 3.3.

2 - 3.3.2 Subtype Declarations

Example: The following example illustrates a clearly readable and understandable mnemonic subtype declaration.

```
--      Weekday_Type is declared in package Basic_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.
```

```
   subtype Weekday_Type is Day_Type range Monday .. Friday;
```

For additional discussion refer to Section 2 - 3. and 2 - 3.3.

2 - 3.3.3 Classification of Operations

No additional Ada-specific impact on TCBs

2 - 3.4 Derived Types

Derived types are subject to the same restrictions as their parent types.

Example: The following example illustrates a clearly readable and understandable mnemonic derived type declaration. The package Key_Manager_Package is declared in Section 2 - 7.4.2.

```
   type Access_Key_Type is new Key_Manager_Package. Key_Type;
   -- the derived subprograms have the following specifications:

   -- procedure Get_Key ( K : out Access_Key_Type );
   -- function "<" ( X, Y : Access_Key_Type ) return BOOLEAN;
   -- function "+" ( X, Y : Access_Key_Type ) return Access_Key_Type;
```

2 - 3.5 Scalar Types

No additional Ada-specific impact on TCBs

2 - 3.5.1 Enumeration Types

The use of enumeration types can improve the readability, understandability, and maintenance of code.

Guideline:

1. An enumeration type should be used to identify a set of discrete items with mnemonic names that promote their readability and understandability.

Example: The following example illustrates a clearly readable and understandable mnemonic enumeration type declaration.

```
--      Security_Classification_Type is declared in
--      package Basic_TCB_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included
--      formally in this appendix.

        type Security_Classification_Type is
             ( Unclassified, Confidential, Secret, Top_Secret );
```

2 - 3.5.2 Character Types

No additional Ada-specific impact on TCBs

2 - 3.5.3 Boolean Types

No additional Ada-specific impact on TCBs

2 - 3.5.4 Integer Types

No additional Ada-specific impact on TCBs

2 - 3.5.5 Operations of Discrete Types

No additional Ada-specific impact on TCBs

2 - 3.5.6 Real Types

No additional Ada-specific impact on TCBs

2 - 3.5.7 Floating Point Types

No additional Ada-specific impact on TCBs

2 - 3.5.8 Operations of Floating Point Types

No additional Ada-specific impact on TCBs

2 - 3.5.9 Fixed Point Types

No additional Ada-specific impact on TCBs

2 - 3.5.10 Operations of Fixed Point Types

No additional Ada-specific impact on TCBs

2 - 3.6 Array Types

Of all data types whose storage is static, array types and their memory management are of primary concern to the software development of TCBs. Static storage is a portion of memory that is set aside at compile time and whose size does not change during program execution. Objects of most Ada data types use static storage. Statically stored data, like that in an array, can generally be accessed more efficiently than data stored dynamically in data structures consisting of access types. This is offset by their potentially inefficient use of memory; this is particularly true of arrays. It would be desirable to include in the next revision of Ada (Ada9x) a predefined pragma that directs Ada compilation to include code that scrubs memory that is local to a subprogram or package just before the scope of the subprogram or package is left during program execution, i.e., just before the visibility of the static (or dynamic) memory is lost. These conflicting needs, efficient execution versus efficient use of memory, are particularly important for large data structures.

Guidelines:

1. Determine whether each data structure should be implemented for efficient execution (static), or for efficient memory usage (dynamic). This determination should be done as early as possible in the development of a TCB system, preferably in the TCB's requirements specification.

2. Monitor the implementation of static storage during design reviews and code walkthroughs. At times the need for a new static data structure will not be identified until design or coding. The need for and use of these new static data structures also should be monitored during design reviews and code walkthroughs.

3. Static storage should be initialized before use and be scrubbed when the data it contains is no longer needed. That is, static storage should contain sensitive data only when the corresponding static data structure is visible during program execution. Thus, local static

storage should be initialized upon entering its scope, and should be scrubbed just before leaving its scope.

Examples: The following examples illustrate clearly readable and understandable mnemonic array type declarations.

```
--     The following declarations in this section are located in the array
--     types version of package Access_Control_List_Types_Package.  Because only
--     essential features of this package need to be shown, it is not included
--     formally in this appendix.

       Max_Number_of_Objects                  : constant NATURAL := 25;
       Max_Number_of_Individuals              : constant NATURAL := 25;
       Max_Number_of_Groups_of_Individuals : constant NATURAL := 25;

       type Named_Objects_List_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Objects ) of
              Basic_TCB_Types_Package. Name_of_Object_Type;    --  see 2 - 4.1

       type Named_Objects_ACL_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Objects ) of
              Named_Object_Record_Type;                         --  see 2 - 3.2.1


       type Named_Individuals_List_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Individuals ) of
              Basic_TCB_Types_Package.
                Name_of_Individual_Type;                        --  see 2 - 4.1

       type Named_Individuals_ACL_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Individuals ) of
              Named_Individual_Record_Type;                     --  see 2 - 3.2.1

       type Groups_of_Named_Individuals_List_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Groups_of_Individuals ) of
              Basic_TCB_Types_Package.
                Name_of_Group_of_Individuals_Type;             -- see 2 - 4.1

       type Groups_of_Named_Individuals_ACL_Type is array
            ( POSITIVE range 1 .. Max_Number_of_Groups_of_Individuals ) of
              Group_of_Named_Individuals_Record_Type;           --  see 2 - 3.2.1
```

2 - 3.6.1 Index Constraints and Discrete Ranges

No additional Ada-specific impact on TCBs

2 - 3.6.2 Operations of Array Types

No additional Ada-specific impact on TCBs

2 - 3.6.3 The Type String

Example: The following example illustrates a clearly readable and understandable mnemonic string type declaration.

```
--      Name_String_Type is declared in package Basic_TCB_Types_Package.
--      Because only essential features of this package need to be shown,
--      it is not included formally in this appendix.

        subtype Name_String_Type is
                STRING ( 1 .. Max_Named_String_Length );     --  see 2 - 3.2.2
```

For additional discussion refer to Sections 2 - 3. and 2 - 3.6.

2 - 3.7 Record Types

All restrictions and implications inherent in the types of a component of a record are implicit for that component.

Guideline:

1. In the declarations of records, assign default values to the record components to avoid reference to undefined variables.

Examples: The following examples illustrate clearly readable and understandable mnemonic record type declarations.

```
--      The following declarations in this section are located in
--      both the access and array types versions of
--      package Access_Control_List_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included formally in
--      this appendix.

        type Named_Object_Record_Type is
          record
            Name : Basic_TCB_Types_Package. Name_of_Object_Type   --  see 2 - 4.1
              := Basic_TCB_Types_Package. Blank_Name_String;    --  see 2 - 3.2.1
```

```
    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;            -- see 3 - 3.1.3

    Authorized_Named_Individuals_List :
               Named_Individuals_List_Type      --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Individuals_List;       --  see 2 - 3.2.1

    Authorized_Groups_of_Named_Individuals_List :
                                                --  see 2 - 3.6 and 2 - 3.8
               Groups_of_Named_Individuals_List_Type
      := Scrubbed_Groups_of_Named_Individuals_List;  --  see 2 - 3.2.1

    Unauthorized_Named_Individuals_List :
                 Named_Individuals_List_Type    --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Individuals_List;       --  see 2 - 3.2.1
end record;


type Named_Individual_Record_Type is
  record
    Name : Basic_TCB_Types_Package.
              Name_of_Individual_Type                    --  see 2 - 4.1
      := Basic_TCB_Types_Package. Blank_Name_String;     --  see 2 - 3.2.1

    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;            -- see 3 - 3.1.3

    Named_Objects_List :
      Named_Objects_List_Type            --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Objects_List;    --  see 2 - 3.2.1
end record;


type Group_of_Named_Individuals_Record_Type is
  record
    Name : Basic_TCB_Types_Package.
              Name_of_Group_of_Individuals_Type         --  see 2 - 4.1
      := Basic_TCB_Types_Package. Blank_Name_String;    --  see 2 - 3.2.1
    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;          -- see 3 - 3.1.3

    Named_Objects_List :
      Named_Objects_List_Type               -- see 2 - 3.6 and 2 - 3.8
        := Scrubbed_Named_Objects_List;     -- see 2 - 3.2.1
end record;
```

2 - 3.7.1 Discriminants

No additional Ada-specific impact on TCBs

2 - 3.7.2 Discriminant Constraints

No additional Ada-specific impact on TCBs

2 - 3.7.3 Variant Parts

No additional Ada-specific impact on TCBs

2 - 3.7.4 Operations of Record Types

No additional Ada-specific impact on TCBs

2 - 3.8 Access Types

The access type in Ada is roughly equivalent to the pointer type in Pascal in that it is used to create and manage dynamic storage.

Dynamic storage is a pool of memory, or heap, that is used for storing data whose demand for memory varies during program execution. It is a useful mechanism that can provide a convenient and flexible means of managing memory when the need for memory is constantly changing.

Access types are used in Ada to implement dynamic storage constructs, like linked lists and queues, which are convenient ways of managing a given system's continually varying demands for memory. "An implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible" [ANSI/MIL-STD-1815A-1983]. In addition, Ada has the **pragma** CONTROLLED, which "specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program" [ANSI/MIL-STD-1815A-1983]. Also, "if an object or one of its subcomponents belongs to a task type, it is considered to be accessible as long as the task is not terminated." Ada's access types can promote an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. Determine whether each data structure should be implemented for efficient memory usage (dynamic), or for efficient execution (static). This determination should be done as early as possible in the development of a TCB system, preferably in the TCB's requirements specification.

2. Avoid using access types and their corresponding objects in areas where aliasing might occur in parameter passing.

3. Avoid using access types in task types. The reason for this is twofold: 1. To prevent the dynamic creation of tasks. 2. The lack of experience with and understanding of passing tasks as parameters in subprogram calls.

4. Monitor the implementation of dynamic storage during preliminary and detailed design reviews and code walkthroughs. At times the need for a new dynamic data structure will not be identified until design or coding. The need for and use of these new dynamic data structures also should be monitored during preliminary and detailed design reviews and code walkthroughs.

5. The TCB's dynamically stored data that is about to be collected by Ada's garbage collection should be protected by deleting the data just before it is collected as garbage. That is, sensitive data that may be accessed must be removed from memory before the memory is deallocated. Also memory should be scrubbed just before it is dynamically allocated with Ada's **new** statement. These additional checks will impinge on system performance.

6. Avoid aliasing that can be introduced with access types. Minimize the number of access type objects that point to any node [ASOS 1987].

7. Avoid using the **pragma** CONTROLLED. Any use of the **pragma** CONTROLLED should be reviewed during design reviews and code walkthroughs.

Examples: The following examples illustrate clearly readable and understandable mnemonic access type declarations. To ensure that the memory allocated to a node is scrubbed, the components of a node's record type should be initialized to scrubbed values, as shown below. Also, ensure that a node record is set to scrub values before it is placed back in the heap, i.e., before nothing points to it any longer.

```
--      The following declarations in this section are located in the access
--      types version of package Access_Control_List_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        type Named_Objects_List_Type;
        type Named_Objects_List_Pointer_Type
          is access Named_Objects_List_Type;

        type Named_Objects_List_Type is
          record
            Name : Basic_TCB_Types_Package.
                     Name_of_Object_Type                    -- see 2 - 4.1
              := Basic_TCB_Types_Package.
                   Blank_Name_String;                       -- see 2 - 3.2.1

            Next : Named_Objects_List_Pointer_Type := null;
        end record;


        type Named_Objects_ACL_Type;
        type Named_Objects_ACL_Pointer_Type
          is access Named_Objects_ACL_Type;

        type Named_Objects_ACL_Type is
          record
            Named_Object_ACL_Record :
              Named_Object_Record_Type
              := Scrubbed_Named_Object_ACL_Record;          -- see 2 - 3.2.1

            Next : Named_Objects_ACL_Pointer_Type
              := null;
        end record;
```

```
type Named_Individuals_List_Type
type Named_Individuals_List_Pointer_Type
  is access Named_Individuals_List_Type;

type Named_Individuals_List_Type is
  record
    Name : Basic_TCB_Types_Package.
             Name_of_Individual_Type                -- see 2 - 4.1
      := Basic_TCB_Types_Package.
           Blank_Name_String;                      -- see 2 - 3.2.1

    Next : Named_Individuals_List_Pointer_Type := null;
end record;


type Named_Individuals_ACL_Type;
type Named_Individuals_ACL_Pointer_Type
  is access Named_Individuals_ACL_Type;

type Named_Individuals_ACL_Type is
  record
    Named_Individual_ACL_Record :
      Named_Individual_Record_Type
    := Scrubbed_Named_Individual_ACL_Record;        -- see 2 - 3.2.1

    Next : Named_Individuals_ACL_Pointer_Type
      := null;
end record;


type Groups_of_Named_Individuals_List_Type
type Groups_of_Named_Individuals_List_Pointer_Type
  is access Groups_of_Named_Individuals_List_Type;

type Groups_of_Named_Individuals_List_Type is
  record
    Name : Basic_TCB_Types_Package.
             Name_of_Group_of_Individuals_Type;     -- see 2 - 4.1
      := Basic_TCB_Types_Package.
           Blank_Name_String;                       -- see 2 - 3.2.1

    Next : Groups_of_Named_Individuals_List_Pointer_Type
      := null;
end record;
```

```
type Groups_of_Named_Individuals_ACL_Type;
type Groups_of_Named_Individuals_ACL_Pointer_Type
  is access Groups_of_Named_Individuals_ACL_Type;

type Groups_of_Named_Individuals_ACL_Type is
  record
    Group_of_Named_Individuals_ACL_Record :
      Group_of_Named_Individuals_Record_Type
      := Scrubbed_Group_of_Named_Individuals_ACL_Record;
                                                  -- see 2 - 3.2.1
    Next : Groups_of_Named_Individuals_ACL_Pointer_Type
      := null;
end record;
```

2 - 3.8.1 Incomplete Type Declarations

No additional Ada-specific impact on TCBs

2 - 3.8.2 Operations of Access Types

No additional Ada-specific impact on TCBs

2 - 3.9 Declarative Parts

No Ada-specific impact on TCBs

2 - 4. Names and Expressions

This chapter of the LRM discusses the use of identifiers as names, combining names into expressions, and rules for evaluation of both names and expressions. The areas of primary interest to the development of TCBs are **Names** (Section 2 - 4.1) and **Allocators** (Section 2 - 4.8).

2 - 4.1 Names

The selection of names can promote the code's readability, understandability, and maintenance.

Guidelines:

1. Name data types and objects with clearly readable and understandable mnemonics.

2. Any data item or group of data should have its own symbolic name that promotes its readability and understandability [ASOS 1987].

Examples: The following examples illustrate clearly readable and understandable mnemonic names.

```
--      The following type declarations in this section are located in
--      package Basic_TCB_Types_Package.  Because only essential features of this
--      package need to be shown, it is not included formally in this appendix.

        subtype Name_of_Object_Type is Name_String_Type;       --  see 2 - 3.2.2

        Name_of_Object : Name_of_Object_Type
          := Blank_Name_String;                                --  see 2 - 3.2.1


        subtype Name_of_Individual_Type is Name_String_Type;   --  see 2 - 3.2.2

        Name_of_Individual : Name_of_Individual_Type
          := Blank_Name_String;                                --  see 2 - 3.2.1


        subtype Name_of_Group_of_Individuals_Type is Name_String_Type;
                                                               --  see 2 - 3.2.2
        Name_of_Group_of_Named_Individuals :
          Name_of_Group_of_Individuals_Type
            := Blank_Name_String;                              --  see 2 - 3.2.1
```

2 - 4.1.1 Index Components

No additional Ada-specific impact on TCBs

2 - 4.1.2 Slices

No additional Ada-specific impact on TCBs

2 - 4.1.3 Selected Components

No additional Ada-specific impact on TCBs

2 - 4.1.4 Attributes

No additional Ada-specific impact on TCBs

2 - 4.2 Literals

No Ada-specific impact on TCBs

2 - 4.3 Aggregates

No Ada-specific impact on TCBs

2 - 4.3.1 Record Aggregates

No Ada-specific impact on TCBs

2 - 4.3.2 Array Aggregates

No Ada-specific impact on TCBs

2 - 4.4 Expressions

No Ada-specific impact on TCBs

2 - 4.5 Operators and Expression Evaluation

No Ada-specific impact on TCBs

2 - 4.5.1 Logical Operators and Short-Circuit Control Forms

No Ada-specific impact on TCBs

2 - 4.5.2 Relational Operators and Membership Tests

No Ada-specific impact on TCBs

2 - 4.5.3 Binary Adding Operators

No Ada-specific impact on TCBs

2 - 4.5.4 Unary Adding Operators

No Ada-specific impact on TCBs

2 - 4.5.5 Multiplying Operators

No Ada-specific impact on TCBs

2 - 4.5.6 Highest Precedence Operators

No Ada-specific impact on TCBs

2 - 4.5.7 Accuracy of Operations with Real Operands

No Ada-specific impact on TCBs

2 - 4.6 Type Conversions

The use of type conversion violates data abstraction, by confusing the meaning of objects whose types are converted.

Guideline:

1. Avoid using type conversion in implementing a TCB system.

2. Security aspects of the use of type conversion should be reviewed during design reviews and code walkthroughs.

2 - 4.7 Qualified Expressions

No Ada-specific impact on TCBs

2 - 4.8 Allocators

Guidelines:

1. Ensure that when using allocators the initialization recommendations (Section 2 - 3.2.1) are taken into consideration. This will prevent programs from operating on objects that are not initialized.

2. Minimize using allocators. Refer to Section 2 - 3.8 for further discussion of access types.

2 - 4.9 Static Expressions and Static Subtypes

No Ada-specific impact on TCBs

2 - 4.10 Universal Expressions

No Ada-specific impact on TCBs

2 - 5. Statements

This chapter of the LRM describes the eight types of Ada statements. These are assignment (with special case for arrays), conditional (if), case, loop, block, exit, return, and goto. For the most part, these are the standard kinds of statements found in most modern day programming languages.

2 - 5.1 Simple and Compound Statements - Sequences of Statements

Guidelines:

1. Statements should satisfy the following criteria: (1) no aliasing, (2) no side effects, and (3) no nonlocal variables. These criteria are good software engineering practice, and as such should not require extra programming effort. These criteria are necessary in that each of them, if not met, may cause an unintended change to be affected during the course of the program execution. These unintended changes may cause an erroneous, or at least unexpected, result.

2. Ada statement semantics should support the general goals of program clarity and unambiguous execution [ASOS 1987].

2 - 5.2 Assignment Statement

No Ada-specific impact on TCBs

2 - 5.2.1 Array Assignments

No Ada-specific impact on TCBs

2 - 5.3 If Statements

In compound if statements, the compiler's order of evaluation could cause side effects.

Guideline:

1. The order of evaluation in compound if statements must be forced by using the short circuit forms, and then and or else.

2 - 5.4 Case Statements

The use of the **others** choice may allow an error that would have been caught at compile time. For example, if the "alternative" expression is an enumeration type that was expanded and if the additional cases were not added to the **case** statement, then the **case** statement would not compile if the **others** choice were not used.

Guideline:
1. Avoid using the **others** choice in **case** statements, particularly, when the "alternative" expression is an enumeration type.

2 - 5.5 **Loop Statements**

No Ada-specific impact on TCBs

2 - 5.6 **Block Statements**

No Ada-specific impact on TCBs

2 - 5.7 **Exit Statements**

No Ada-specific impact on TCBs

2 - 5.8 **Return Statements**

The leaving of a subprogram or **accept** statement needs to be clearly traceable to a single **return** statement.

Guideline:
1. All subprograms and **accept** statements should have a single **return** statement for leaving their scope of execution.

2 - 5.9 Goto Statements

Goto statements provide the means for a program to transfer control, in an unstructured manner, of its operation elsewhere in the program. The larger the module, the more potential problems that **goto** statements can present. In particular, undesired redirection of program execution may be inserted that is hard to track. Because this transfer of control may be unregulated, ensuring the security of a TCB system is very difficult. Also, Ada provides more structured control structures (e.g., **for** and **while** loops) that make using **goto** statements unnecessary.

Guidelines:

1. Avoid using Goto statements in any system implementation, especially that of a TCB.

2. Security aspects of proposed **goto** statements should be addressed as early as possible in the development of a TCB system, preferably in its preliminary and detailed design phase.

3. Security aspects of all **goto** statements should be reviewed during design reviews and code walkthroughs.

2 - 6. Subprograms

This chapter of the LRM defines the mechanism for describing subprograms (technically, procedures and functions), the mechanisms for their invocation, and the manner of parameter passing. In the absence of concurrency, as with other aspects of the Ada language, most aspects of subprogram declaration and invocation are equivalent with respect to other languages. Some specific differences that are relevant to Ada are the issues of subprogram declarations (Section 2 - 6.1), formal parameter modes (Section 2 - 6.2), and parameter "aliasing" (Section 2 - 6.4).

Guidelines:

1. Subprograms should have only one entry point and only one exit point to promote testing and maintaining the subprograms.

2. Avoid side effects in subprograms, e.g., on global data (non-parameter data), by reference to as well as modifying the global data within a subprogram [ASOS 1987].

3. Avoid creating large subprograms because thoroughly testing large subprograms is more difficult than testing small subprograms. That is, avoid creating subprograms with a larger number of possible logical paths (i.e., transfers of control of program execution), which is typical in large subprograms. The use of a large subprogram should be justified in the preliminary and detailed design reviews and code walkthroughs.

4. No subprogram, and especially no large subprogram, should contain goto statements. The use of a goto statement should be justified in preliminary and detailed design reviews and code walkthroughs.

2 - 6.1 Subprogram Declarations

Equivalently named subprograms with the same parameter type profile are allowed because the scoping rules in Ada clearly define which subprogram is visible at any point in the program text.

Guideline:

1. Justify equivalently named subprograms from a human viewpoint because the additional complexity of overloaded names could cause difficulty in the software development process. The equivalent naming of subprograms should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 6.2 Formal Parameter Modes

Guideline:

1. Include the definition of the default values for **out** parameters, akin to that described in Section 2 - 3.2.1 for object declarations. Doing so helps to prevent erroneous programs.

2 - 6.3 Subprogram Bodies

Data present in the data objects declared within a subprogram remains in the memory containing the subprogram's activation record after program execution has left the subprogram. Therefore the data should be scrubbed from the memory just before the scope of the subprogram is left.

Guideline:

1. Scrub data present in the data objects declared within subprograms before returning from a subprogram's call.

2 - 6.3.1 Conformance Rules

No additional Ada-specific impact on TCBs

2 - 6.3.2 Inline Expansion of Subprograms

No additional Ada-specific impact on TCBs

2 - 6.4 Subprogram Calls

No additional Ada-specific impact on TCBs

2 - 6.4.1 Parameter Associations

Guideline:

1. Aliasing across subprogram boundaries must be minimized. Harmful aliasing may occur when formal parameters are assumed to represent distinct variables, but actual parameters supplied are not distinct [ASOS 1987].

2 - 6.4.2 Default Parameters

Guidelines:

1. Use named parameter association to improve readability and understandability of subprogram calls.

Example: This example illustrates named parameter association.

```
procedure Search_File ( File : in out Access_Control_List_Types_Package.
                                      ACL_File_Type;
                        Key   : in     Name;
                        Index :    out File_Index );

Search_File ( File  => Named_Objects_Access_Control_List_Manager_Package.
                       ACL_File,
              Key   => "Smith , J",
              Index => Record_Entry );
```

2. Default parameters may be used when a subprogram has parameters whose actual values do not change over most calls.

Example: This example from package TEXT_IO illustrates default parameters.

```
procedure CREATE ( FILE : in out FILE_TYPE;
                   MODE : in     FILE_MODE := OUT_FILE;
                   NAME : in     STRING    := "";
                   FORM : in     STRING    := "" );

TEXT_IO. CREATE ( FILE => Named_Objects_Access_Control_List_Manager_Package.
                          ACL_File );
```

2 - 6.5 Function Subprograms

Function subprograms should be truly functional, which is determined by the following two guidelines.

Guidelines:

1. Prohibit functions from containing either input or output operations (Chapter 14 of the LRM) to be consistent with the fundamental characteristics of functions, i.e., to return only a single value [ASOS 1987].

2. Function subprograms should not reference any nonlocal (i.e., global) variables.

2 - 6.6 Parameter and Result Type Profile - Overloading of Subprograms

For the discussion of overloading refer to Section 2 - 6.1.

2 - 6.7 Overloading of Operators

For the discussion of overloading refer to Section 2 - 6.1.

2 - 7. Packages

This chapter of the LRM discusses with the specification of packages as a means to encapsulate data and subprograms into a single structure. The use of packages directly supports the notion of abstract data types, a common mechanism in software engineering used to reduce program complexity. The following software engineering principles are discussed in this section: data abstraction, information hiding, modularity, and localization. Compilation of specifications is discussed in **Package Specifications and Declarations** (Section 2 - 7.2).

Ada's data abstraction mechanisms are well suited to represent the data objects in a system's design, namely, that of a TCB. They serve the conceptual manipulation of the data objects in a relatively high-level of abstraction without regard to their underlying representation. Ada allows data to be abstracted with abstract data types. An abstract data type is a construct that "denotes a class of objects whose behavior is defined by a set of values and a set of operations" [Booch 1987A, p.613]. Ada's two primary features that promote the creation of abstract data types are its strong data typing facilities (which is discussed above) and its packaging mechanism. A package can be used to define an abstract data type by encapsulating its underlying data types and the operations associated with the abstract data type. Using Ada's data abstraction techniques aids the representation of data objects in the problem space of any system, including a TCB. "Abstraction aids in the maintainability and understandability of systems by reducing the details a developer needs to know at any given level" [Booch 1987B, p.33]. This sound software engineering technique can promote an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. Use packages to enforce visibility rules of data accessibility [ASOS 1987].

2. Use Ada's data abstraction mechanisms to represent data objects in a TCB system's design. Identify and implement abstract data types in the design by encapsulating them in packages. That is, the package specification should contain an abstract data type declared as a (limited) private type, and those operations that can be performed on objects of this abstract data type. No direct access to data structures encapsulated by such packages should be allowed, i.e., access to these data structures should be performed indirectly through the operations provided in the package specification.

3. Data objects should be created using Ada's strong data typing facilities and its packaging mechanism, which promote the

understandability, maintainability, and reusability of a system's design and code.

4. **The fundamental data abstractions should be identified early, preferably during the TCB system's requirements specification. Their implementation should be monitored during preliminary and detailed design reviews and code walkthroughs.**

Ada's information hiding facilities complement its data abstraction capabilities. Whereas abstractions extract the essential details of a given level, "the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system" [Ross, Goodenough, and Irvine 1975, p.67]. "Information hiding therefore suppresses how an object or operation is implemented, and so focuses our attention on the higher abstraction" [Booch 1987B, p.33]. Two Ada constructs that are well suited for implementing information hiding are packages and private types. "Packages can be used to hide information from the rest of the program while making explicit the interface with other program parts. This has the advantage that implementation details of each package can be changed by altering only its body, and that the rest of the program may be understood without reference to these details" [Nissen and Wallis 1984, p.122].

Hiding the information about the implementation of the data abstraction of a data object is achieved in Ada by encapsulating the abstraction in a package, i.e., by hiding the implementation of the object and controlling access to the object so as to encourage and enforce the abstraction. This typically is done with the use of private types and limited private types. In particular, Ada's private types enable the focus to be placed on higher-level real-world abstractions rather than on the details of an implementation. The following implicit operations may be performed on private types: assignment, tests of equality and inequality, explicit type conversion, membership tests, type qualification, and the use of selected components for the selection of any of the private type's discriminant. For limited private types, though, only those operations defined in the corresponding package specification are allowed. "Private types prevent misuse of structures by users, presenting them only with the abstract operations appropriate for the abstractions involved" [Nissen and Wallis 1984, p.131]. For more detailed information on private types, refer to the Ada language reference manual [ANSI/MIL-STD-1815A-1983].

An example of using Ada's information hiding (and data abstraction) would be to implement a package that defined a linked list structure. Only those details required by a user of the linked list package would be provided in the package specification (data abstraction), e.g., the operations allowed on the linked list. The implementation, though, of the linked list would be hidden inside the package body. The user of the package does not need to know how the linked list is implemented; therefore, information on its implementation is hidden from the user.

The understandability of systems are enhanced "when, at each level of abstraction, we permit only certain operations and prevent any operations that violate our logical view of that level" [Booch 1987B, p.33]. Thus, this sound software engineering technique can promote an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. As much as possible of the implementation detail should be hidden in the body of the package that corresponds to an abstract data type. That is, Ada's information hiding facilities should be used to complement its data abstraction capabilities by making inaccessible certain details that should not affect (i.e., be visible to) other parts of a system. The use of information hiding should serve to minimize the compromising of the system's software design and structure. That is, it should localize logically related implementation details of the abstract data type, and thus minimize coupling and maximize cohesion in the system.

2. Information hiding should be implemented with the two Ada constructs, packages and private types. This implementation should be monitored during preliminary and detailed design reviews, and code walkthroughs.

3. The underlying data structures that constitute an object of an abstract data type should not be directly accessible. That is, the data structures should be accessible only indirectly through the subprograms specified in the object's package specification, that define the operations available to be performed on the object.

Modularity provides the mechanism for collecting logically related abstractions. It is used to create the structure of an object that makes the attainment of some purpose easier. Modularity is purposeful structuring, which is usually achieved in a large system by decomposing the system top-down with modules that are either functional (procedure-oriented) or declarative (object-oriented) [Booch 1987B, p.34]. It is composed preferably of existing reusable bottom-up software components. This structuring should be performed to minimize the coupling between modules (i.e., minimizing dependencies between modules), and to strengthen the cohesion within modules (i.e., the components of a given module are functionally and logically dependent) [Booch 1987B, p.34].

Localization is the collecting of logically related computational resources in one physical module that is sufficiently independent of other modules. Localization thus helps to create modules that exhibit loose coupling and strong cohesion.

The principles of modularity and localization directly support modifiability and understandability [Booch 1987B, p.34]. Any given module should be understandable and relatively independent of other modules. Design decisions localized in given modules limit the effects of a modification to a small set of modules. Thus, the use of modularization that limits the interconnections among program modules, and the localizing of logically related resources into modules are sound software engineering techniques that can promote an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. Name packages and subprograms with clearly understandable and readable mnemonics.

2. Use modularization and localization to create packages and their subprograms so that they exhibit loose coupling between the subprograms and packages, and exhibit strong cohesion within their respective implementations. In contrast to modularizing, logically related computational resources should be localized by collecting them into a package and its subprograms. Localize design decisions in packages and subprograms to minimize rippling side effects of a modification to a small set of modules.

3. Logically related data abstractions should be collected into a class of packages.

4. Modularity and localization should be evaluated during preliminary and detailed design reviews and code walkthroughs.

2 - 7.1 Package Structure

No additional Ada-specific impact on TCBs

2 - 7.2 Package Specifications and Declarations

Ada provides the ability to compile package specifications during the design stage of system development. This aids in the early checking of the system requirement specifications and design, e.g., checking the relationships and interactions between modules before system development progresses into the coding stage. Thus, the quality of the requirement specifications and design is promoted by using package specification compilation.

Guidelines:

1. Package specifications should be compiled during the design stage of Ada system development to check the consistency and quality of the design.

2. The system requirement specifications and design process should use this capability to check the relationships and interactions between modules before system development progresses into the coding stage.

3. In package specification documentation include the specification of the effects produced by each subprogram defined in the package [ASOS 1987].

Example: This example illustrates data abstraction, information hiding, modularity, and localization achieved with package specification and declarations.

```
with Basic_TCB_Types_Package;
with Access_Control_List_Types_Package;
package Named_Objects_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_Named_Object_ACL_Record
      ( Name_of_Object : in     Basic_TCB_Types_Package.
                                  Name_of_Object_Type;    -- see 2 - 4.1
        Named_Object_ACL_Record :  out
          Access_Control_List_Types_Package.
            Named_Object_Record_Type );                   -- see 2 - 4.1

    procedure Insert_Named_Object_ACL_Record
      ( Named_Object_ACL_Record : in
          Access_Control_List_Types_Package.
            Named_Object_Record_Type );                   -- see 2 - 4.1

    procedure Delete_Named_Object_ACL_Record
      ( Name_of_Object : in Basic_TCB_Types_Package.
                              Name_of_Object_Type );      -- see 2 - 4.1

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;
```

```
private

   -- . . .

end Named_Objects_Access_Control_List_Manager_Package;


package body Named_Objects_Access_Control_List_Manager_Package is

   -- By declaring the Named_Objects_ACL (see 2 - 12.4) in the
   -- body of this package rather than in the specification,
   -- it is hidden from the user of this package.  Thus, the user
   -- can gain only indirect access to it through the subprograms
   -- declared in the specification.  The typical list manipulation
   -- operations ( e.g., as illustrated by Booch 1987A and
   -- Feldman 1985 ) are only provided in the package body.

   -- . . .

   -- Typical list manipulation operations

   -- . . .

   procedure Get_Named_Object_ACL_Record
     ( Name_of_Object : in     Basic_TCB_Types_Package.
                                 Name_of_Object_Type;     --   see 2 - 4.1
       Named_Object_ACL_Record :   out
         Access_Control_List_Types_Package.
           Named_Object_Record_Type ) is                  --   see 2 - 4.1

      -- . . .

    begin  -- Get_Named_Object_ACL_Record

      -- . . .

      -- Sequence through the Named_Object_ACL using the typical
      -- list manipulation operations to locate and get the
      -- indicated Named_Object_ACL_Record.

      -- . . .

   end Get_Named_Object_ACL_Record;
```

```
procedure Insert_Named_Object_ACL_Record
  ( Named_Object_ACL_Record : in
      Access_Control_List_Types_Package.
        Named_Object_Record_Type ) is                 -- see 2 - 4.1

    -- . . .

  begin  -- Insert_Named_Object_ACL_Record

    -- . . .

    -- Sequence through the Named_Object_ACL using the typical
    -- list manipulation operations to locate the appropriate
    -- place to insert the indicated Named_Object_ACL_Record.
    -- This location is determined by a predefined mechanism,
    -- e.g., alphabetizing by the Named_Object_ACL_Record.Name,
    -- or more crudely by a simple (FIFO) stack or (FILO) queue.

    -- . . .

end Insert_Named_Object_ACL_Record;

procedure Delete_Named_Object_ACL_Record
  ( Name_of_Object : in Basic_TCB_Types_Package.
                          Name_of_Object_Type ) is    -- see 2 - 4.1

    -- . . .

  begin  -- Delete_Named_Object_ACL_Record

    -- . . .

    -- Sequence through the Named_Object_ACL using the typical
    -- list manipulation operations to locate, scrub, and delete
    -- the indicated Named_Object_ACL_Record.

    -- . . .

end Delete_Named_Object_ACL_Record;

  -- . . .

end Named_Objects_Access_Control_List_Manager_Package;
```

Global variables are a convenient means of passing data between different parts of a system. "Global variables" may be implemented in Ada in a package specification or a subprogram specification. The primary advantage of using global variables is that some efficiency in data transfer within the program is typically introduced. This advantage is almost always offset by the problems caused by using global variables.

Guidelines:

1. The use of global variables should be avoided because they cause difficulty in tracing accesses of and modifications to the variables.

2. Security aspects of proposed global variables should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

3. Security aspects of all global variables should be reviewed during preliminary and detailed design reviews and code walkthroughs.

2 - 7.3 Package Bodies

No additional Ada-specific impact on TCBs

2 - 7.4 Private Type and Deferred Constant Declarations

Refer to the discussions of data abstraction and information iding in Section 2 - 7, Packages.

2 - 7.4.1 Private Types

For the discussion of the use of private types in packages, refer to Section 2 - 7.

2 - 7.4.2 Operations of a Private Type

Example: The abstract data type, Key_Type, is created with the use of the package Key_Manager_Package and its private type, Key_Type.

```
package Key_Manager_Package is
    type Key_Type is private;
    Null_Key : constant Key_Type;
    procedure Get_Key ( K : out Key_Type );
    function "<" ( X, Y : Key_Type ) return BOOLEAN;
    function "+" ( X, Y : Key_Type ) return Key_Type;

  private
    type Key_Type is new NATURAL;
    Null_Key : constant Key_Type := 0;
end Key_Manager_Package;


package body Key_Manager_Package is

    Last_Key : Key_Type := 0;

    procedure Get_Key ( K : out Key_Type ) is
      begin
        Last_Key := Last_Key + 1;
        K := Last_Key;
    end Get_Key;

    function "<" ( X, Y : Key_Type ) return BOOLEAN is
      begin
        return NATURAL ( X ) < NATURAL ( Y );
    end "<";

    function "+" ( X, Y : Key_Type ) return Key_Type is
      begin
        return Key_Type ( NATURAL ( X ) + NATURAL ( Y ) );
    end "+";

end Key_Manager_Package;
```

For additional discussion refer to section 2 - 7.

2 - 7.4.3 Deferred Constants

No additional Ada-specific impact on TCBs

2 - 7.4.4 Limited Types

No additional Ada-specific impact on TCBs

2 - 7.5 Example of a Table Management Package

No additional Ada-specific impact on TCBs

2 - 7.6 Example of a Text Handling Package

No additional Ada-specific impact on TCBs

2 - 8. Visibility Rules

This chapter of the LRM establishes rules for determining the visibility of names and identifiers in the Ada program text. For the most part, such rules are applicable at the syntactic and semantic phases of the analysis of the Ada program text. The two sections of primary concern to software development of TCBs are Use Clauses (Section 2 - 8.4) and Renaming Declarations (Section 2 - 8.5).

2 - 8.1 Declarative Region

No Ada-specific impact on TCBs

2 - 8.2 Scope of Declarations

No Ada-specific impact on TCBs

2 - 8.3 Visibility

Guidelines:

1. Avoid using global variables to minimize side effects, but when they must be used, restrict the visibility of global data through the scoping mechanism [ASOS 1987].

2. Avoid the nesting of subprogram declarations, so as to eliminate the potential for the usage of global data other than that defined in the main program [ASOS 1987].

2 - 8.4 Use Clauses

Ada's "use clause achieves direct visibility of declarations that appear in the visible parts of named packages." The use of the use clause makes identifying the origin of an invoked subprogram difficult. The impact on testing is that the specification and definition of the invoked subprogram are not accessible to a tester. Also, a similar effect, which is equally adverse, occurs in maintenance when it is difficult to identify the origin of the invoked subprogram.

Guideline:

1. Avoid using use clauses in Ada TCB system implementation.

2 - 8.5 Renaming Declarations

Ada's "renaming declaration declares another name for an entity." The use of renaming makes identifying the origin of an invoked subprogram difficult. The impact on testing is that the specification and definition of the invoked subprogram are not accessible to a tester. Also, a similar effect, which is equally adverse, occurs in maintenance when it is difficult to identify the origin of the invoked subprogram.

Guideline:

1. Avoid using renaming declarations.

2. Limit the redefinition of operators. At least limit the scope in which an operator is redefined [ASOS 1987].

2 - 8.6 The Package Standard

No Ada-specific impact on TCBs

2 - 8.7 The Context of Overload Resolution

No Ada-specific impact on TCBs

2 - 9. Tasks

This chapter of the LRM discusses the concurrent programming aspects of the Ada language. Concurrent programming with tasking is discussed in this section. The discussion on system timing from package CALENDAR is located in **Delay Statements, Duration, and Time** (Section 2 - 9.6). Also of interest are Sections 2 - 9.1 **Task Specifications and Task Bodies**, 2 - 9.9 **Task and Entry Attributes**, and 2 - 9.11 **Shared Variables**.

In contrast to other languages, Ada incorporates its concurrent programming mechanism, tasking, as an integral part of its definition. Concurrent processes, in particular tasks, are processes that may execute in parallel on multiple processors or independently scheduled processes on a single processor. That is, they involve the simultaneous, or timeshared, execution of processes. Tasks may interact with each other, and one task may suspend execution pending receipt of information from another task or the occurrence of an external event. Despite the problems associated with concurrent programming and Ada's tasking, it is a useful mechanism that likely is required for the effective implementation of a TCB system. Thus, Ada's tasking can promote an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. Minimize the use of tasking [ASOS 1987].

2. Manage communication between tasks in a TCB system with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels), so as to prevent the introduction of covert channels. Communications between tasks (e.g., rendezvous) should be logged in the audit trail.

3. Tasking implementations should avoid using access types (for dynamic storage) and global and shared variables.

4. Management of intertask communications should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

5. Implementation of these intertask communications should be monitored during the TCB system's preliminary and detailed design reviews and code walkthroughs.

2 - 9.1 Task Specifications and Task Bodies

The compilation of task specifications offers the same benefits as the compilation of package specifications which were discussed in Section 2-7.2.

2 - 9.2 Task Types and Task Objects

Management of intertask communications is a very important aspect of the use of tasks. The use of semaphores for this purpose is presented in the following example.

Example: This example illustrates trusted generalized mechanisms to control and regulate intertask communications with semaphores.

```
with Mandatory_Access_Control_Types_Package;         -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;       -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    -- see 3 - 3.2.3
generic

  Max_Number_of_Tasks_Allowed : in NATURAL := 1;

package Generic_Counting_Semaphore_Manager_Package is

  task type Counting_Semaphore_Task_Type is
    entry Allow_Task_to_Pass (
            Other_Task_MAC_Record : in
              Mandatory_Access_Control_Types_Package.
                MAC_Record_Type );                   -- see 3 - 3.1.3

    entry Release_Task (
            Other_Task_MAC_Record : in
              Mandatory_Access_Control_Types_Package.
                MAC_Record_Type );                   -- see 3 - 3.1.3

  end Counting_Semaphore_Task_Type;

  -- . . .

end Generic_Counting_Semaphore_Manager_Package;
```

```
package body Generic_Counting_Semaphore_Manager_Package is

  task body Counting_Semaphore_Task_Type is

      Number_of_Tasks : INTEGER := Max_Number_of_Tasks_Allowed;

      Local_MAC_Record :
        Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
          MAC_Record_Type;

      Other_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
          MAC_Record_Type;

    begin   -- Counting_Semaphore_Task_Type
      loop
        select
            when Number_of_Tasks > 0 =>
              accept Allow_Task_to_Pass (
                       Other_Task_MAC_Record : in
                         Mandatory_Access_Control_Types_Package.
                           MAC_Record_Type ) do      --  see 3 - 3.1.3

                if Mandatory_Access_Control_Manager_Package.
                     Sensitivity_Labels_Match        --  see 3 - 3.1.3
                       ( Local_MAC_Record. Sensitivity_Label,
                         Other_Task_MAC_Record.
                           Sensitivity_Label ) then

                  Audit_Trail_Manager_Package.       --  see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_MAC_Record, Other_Task_MAC_Record );

                  Number_of_Tasks := Number_of_Tasks - 1;
                end if;
              end Allow_Task_to_Pass;

          or
            when Number_of_Tasks < Max_Number_of_Tasks_Allowed
              => accept Release_Task (
                          Other_Task_MAC_Record : in
                            Mandatory_Access_Control_Types_Package.
                              MAC_Record_Type ) do  --  see 3 - 3.1.3
                    if Mandatory_Access_Control_Manager_Package.
                         Sensitivity_Labels_Match      --  see 3 - 3.1.3
                           ( Local_MAC_Record. Sensitivity_Label,
                             Other_Task_MAC_Record.
                               Sensitivity_Label ) then

                      Audit_Trail_Manager_Package.     --  see 3 - 3.2.3
                        Log_Subjects_Access_to_Object_in_Audit_Trail (
                          Local_MAC_Record, Other_Task_MAC_Record );

                      Number_of_Tasks := Number_of_Tasks + 1;
                    end if;
                  end Release_Task;
            end select;
          end loop;
        end Counting_Semaphore_Task_Type;

        -- . . .

      end Generic_Counting_Semaphore_Manager_Package;
```

2 - 9.3 Task Execution - Task Activation

For additional discussion of tasking refer to Section 2 - 9.

Guideline:

1. Initialize all object declarations in task bodies. This includes components of record types.

2 - 9.4 Task Dependence - Termination of Tasks

For additional discussion of tasking refer to Section 2 - 9.

Guideline:

1. Scrub all objects declared in a task before terminating the task.

2 - 9.5 Entries, Entry Calls, and Accept Statements

Guideline:

1. Ensure that entry calls and their corresponding accept statements are monitored by checking the sensitivity labels associated with the respective tasks involved in a given rendezvous. Also, the rendezvous

should be logged in the audit trail. For additional tasking guidelines refer to Section 2 - 9.0. Examples illustrating this guideline are in Sections 2 - 9.2 and 2 - 9.12.

2 - 9.6 Delay Statements, Duration, and Time

Ada provides access to system timing through the **package** CALENDAR. Only system timing is available from **package** CALENDAR. To get timing from an external clock a new package must be developed, probably with representation clauses (more fully detailed in Chapter 13), which introduce their own complications.

Guidelines:

1. Use of Ada's **package** CALENDAR should be minimized in Ada TCB system implementation; it will probably be required for time stamping.

2. Security aspects of using **package** CALENDAR should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications. In particular, account for the possible failure of the system timing available from **package** CALENDAR to correspond precisely to the system clock.

3. Security aspects of using **package** CALENDAR should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 9.7 Select Statements

2 - 9.7.1 Selective Waits

Guideline:

1. Avoid the use of selective waits: they introduce indeterminary. The introduction of indeterminacy, especially with selective waits, should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 9.7.2 Conditional Entry Calls

No additional Ada-specific impact on TCBs

2 - 9.7.3 Timed Entry Calls

No additional Ada-specific impact on TCBs

2 - 9.8 Priorities

No additional Ada-specific impact on TCBs

2 - 9.9 Task and Entry Attributes

Tasks and entries have three attributes as specified in the LRM: T'CALLABLE, T'TERMINATED, and E'COUNT. The use of these dynamic attributes enables the passing of information in a manner which is much more difficult to keep track of than the normal manner of parameter passing.

Guideline:
1. Avoid using task and entry attributes. If used, the use should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 9.10 Abort Statements

For additional discussion of tasking refer to Section 2 - 9.

Guideline:
1. Scrub all objects declared in a task before aborting the task.

2 - 9.11 Shared Variables

Shared variables are the major construct in tasking that will have to be restricted (although perhaps simulated through use of other constructs using synchronization) in the software development of TCBs.

Guidelines:
1. Avoid using shared variables because they cause difficulty in tracing accesses of and modifications to the variables.

2. Security aspects of proposed shared variables should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

3. Security aspects of all shared variables should be reviewed during preliminary and detailed design reviews and code walkthroughs.

2 - 9.12 Example of Tasking

Example: This example illustrates trusted generalized mechanisms to control and regulate intertask communications with mailboxes.

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
generic

  type Message_Type is private;

  Max_Number_of_Messages : in NATURAL := 24;

package Generic_Mailbox_Manager_Package is

  procedure Send    ( Message          : in  Message_Type;
                      Local_MAC_Record : in
                        Mandatory_Access_Control_Types_Package.
                          MAC_Record_Type );           -- see 3 - 3.1.3

  procedure Receive ( Message          :    out Message_Type;
                      Local_MAC_Record : in
                        Mandatory_Access_Control_Types_Package.
                          MAC_Record_Type );           -- see 3 - 3.1.3

end Generic_Mailbox_Manager_Package;


with Mandatory_Access_Control_Manager_Package;         -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                      -- see 3 - 3.2.3
package body Generic_Mailbox_Manager_Package is

  task Manager_Task is

    entry Deposit ( Message               : in Message_Type;
                    Other_Task_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type );             -- see 3 - 3.1.3

    entry Remove  ( Message               :    out Message_Type;
                    Other_Task_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type );             -- see 3 - 3.1.3
  end Manager_Task;
```

```
procedure Send ( Message          : in  Message_Type;
                 Local_MAC_Record : in
                   Mandatory_Access_Control_Types_Package.
                     MAC_Record_Type ) is         --  see 3 - 3.1.3
  begin
    Manager_Task. Deposit ( Message, Local_MAC_Record );
end Send;

procedure Receive ( Message          :   out Message_Type;
                    Local_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type ) is      --  see 3 - 3.1.3
  begin
    Manager_Task. Remove ( Message, Local_MAC_Record );
end Receive;


task body Manager_Task is

    subtype Mailbox_Slot_Index_Type is INTEGER range
      0 .. ( Max_Number_of_Messages - 1 );

    Head_Slot : Mailbox_Slot_Index_Type := 0;
    Tail_Slot : Mailbox_Slot_Index_Type := 0;

    Message_Number : INTEGER range 0 .. Max_Number_of_Messages;

    Mailbox : array ( Mailbox_Slot_Index_Type ) of Message_Type;

    Local_MAC_Record :
      Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
        MAC_Record_Type;

    Other_Task_MAC_Record :
      Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
        MAC_Record_Type;

  begin
    loop
      select
          when Message_Number < Max_Number_of_Messages =>
            accept Deposit ( Message                 : in Message_Type;
                             Other_Task_MAC_Record : in
                               Mandatory_Access_Control_Types_Package.
                                 MAC_Record_Type ) do
                                                   --  see 3 - 3.1.3
          if Mandatory_Access_Control_Manager_Package.
               Sensitivity_Labels_Match          -- see 3 - 3.1.3
                 ( Local_MAC_Record. Sensitivity_Label,
                   Other_Task_MAC_Record.
                     Sensitivity_Label ) then

            Audit_Trail_Manager_Package.          -- see 3 - 3.2.3
              Log_Subjects_Access_to_Object_in_Audit_Trail (
                Local_MAC_Record, Other_Task_MAC_Record );

            Mailbox ( Head_Slot ) := Message;

            Head_Slot := ( Head_Slot + 1 ) mod
                         Max_Number_of_Messages;

            Message_Number := Message_Number + 1;
          end if;
        end Deposit;

    or
      when Message_Number > 0 =>
        accept Remove ( Message               : out Message_Type;
                        Other_Task_MAC_Record : in
                          Mandatory_Access_Control_Types_Package.
                            MAC_Record_Type ) do
                                                  -- see 3 - 3.1.3

          if Mandatory_Access_Control_Manager_Package.
               Sensitivity_Labels_Match          -- see 3 - 3.1.3
                 ( Local_MAC_Record. Sensitivity_Label,
                   Other_Task_MAC_Record.
                     Sensitivity_Label ) then

            Audit_Trail_Manager_Package.          -- see 3 - 3.2.3
              Log_Subjects_Access_to_Object_in_Audit_Trail (
                Local_MAC_Record, Other_Task_MAC_Record );

            Message := Mailbox ( Head_Slot );

            Tail_Slot := ( Tail_Slot + 1 ) mod
                         Max_Number_of_Messages;
```

```
                    Message_Number := Message_Number - 1;
                  end if;
                end Remove;
          end select;
        end loop;
  end Manager_Task;

end Generic_Mailbox_Manager_Package;
```

2 - 10. Program Structure and Compilation Issues

This chapter of the LRM describes the units of compilation, attends to the ordering requirements for program libraries, and touches briefly on the results of optimizations. Reusable code is discussed in Section 2 - 10.4 **The Program Library**.

2 - 10.1 Compilation Units - Library Units

Guideline:

1. Use libraries to greatly facilitate good configuration management [ASOS 1987].

2 - 10.1.1 Context Clauses - With Clauses

No Ada-specific impact on TCBs

2 - 10.1.2 Examples of Compilation Units

No Ada-specific impact on TCBs

2 - 10.2 Subunits of Compilation Units

2 - 10.2.1 Examples of Subunits

No Ada-specific impact on TCBs

2 - 10.3 Order of Compilation

No Ada-specific impact on TCBs

2 - 10.4 The Program Library

The program library should contain reusable code. The implementation of an Ada TCB system can be aided with the reuse of evaluated code that has been demonstrated to sufficiently satisfy the security class of the given TCB. Ada's generic units are helpful in creating reusable code. "Generics provide a powerful means by which a program may be 'factorized' in order to shorten code, and reduce incidence of errors, by avoiding redefining items which appear in several places in the program" [Nissen and Wallis 1984, p.181]. When code is reused, the number of errors in the code is reduced, because errors in such code are fixed when they are identified. Additional time and effort is required during the design phase of developing reusable code for a TCB system. The additional time and effort will pay for itself when the resulting reusable code is used in multiple instances in the current TCB and future TCBs. Thus, the reuse of evaluated code is a sound software engineering technique that can promote the efficient production of an Ada TCB system's design, implementation, and maintenance.

Guidelines:

1. Use existing evaluated reusable software components as much as possible to create modular software to promote efficient Ada-TCB system's design, implementation, and maintenance. That is, reuse the code and/or the design that has been demonstrated to sufficiently satisfy the security class of the given Ada TCB system.

2. Create, manage, and use libraries of evaluated reusable software components. Libraries of evaluated reusable Ada source code should be established and managed by the security administrator, who supervises access to and use of the libraries. An example of such a library is discussed in the paper "A Secure SDS Software Library" [Hadley, et. al. 1987].

3. Ada's generic units should be taken full advantage of when creating this reusable code, as exemplified in Booch's Software Components with Ada [1987A].

4. Monitor the implementation of these libraries of evaluated reusable software components during preliminary and detailed design reviews and code walkthroughs.

5. Errors found in code taken from libraries of evaluated reusable software components should be reported to the library manager.

2 - 10.5 Elaboration of Library Units

No Ada-specific impact on TCBs

2 - 10.6 Program Optimization

No Ada-specific impact on TCBs

2 - 11. Exceptions

This chapter of the LRM defines the exception constructs, and mechanisms, and rules of handling exceptions within programs. A general discussion of exceptions is located in this section. Other sections of interest are 2-11.4 **Exception Handling**, 2 - 11.4.1 **Exceptions Raised During the Exception of Statements**, and 2 - 11.7 **Suppressing Checks**.

Ada handles program execution errors or other exceptional situations with its exception handling mechanism. An exception may be used to alter control of program execution (e.g., handling the exception outside a subprogram or a task where the exception was raised).

Using **pragma** SUPPRESS prevents the raising of exceptions for selected checks, which can serve to monitor the proper execution of the program during runtime. Ada code generated when using **pragma** SUPPRESS cannot be trusted to work as expected because Ada compilers currently are not validated when **pragma** SUPPRESS is used.

Guidelines:

1. Exception handling must be managed in a TCB system. Handling an exception between the point at which the exception is raised and the place where it is handled (especially if they are in separate modules) must be enforced in a TCB system with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). Each exception should be labeled so that the initiator of the exception is known by its exception handler.

2. Only define exceptions to handle events that occur infrequently [ASOS 1987].

3. Handle frequent occurrences of bad data by direct coding in subprograms [ASOS 1987].

4. Security aspects of the various exception handling operations should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

5. These operations should be stipulated by the TCB system requirements specification or design documents.

6. Implementation of these operations should be managed during preliminary and detailed design reviews and code walkthroughs.

7. Avoid using **pragma** SUPPRESS.

Example: This example illustrates trusted exception handling that allows program execution to continue in a trusted manner.

```
generic

  -- . . .

  type Name_Type is private;
  type ACL_Record_Type is private;

  -- . . .

package Generic_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in      Name_Type;
        ACL_Record :     out ACL_Record_Type );

    procedure Insert_ACL_Record
      ( ACL_Record : in ACL_Record_Type  );

    procedure Delete_ACL_Record
      ( Name : in Name_Type );

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;

  private

    -- . . .

end Generic_Access_Control_List_Manager_Package;


with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;        -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                     -- see 3 - 3.2.3
package body Generic_Access_Control_List_Manager_Package is
```

```
-- The user can gain only indirect access to instantiated
-- access control list through the subprograms declared in the
-- package specification.  Thus the access control list data
-- structure is hidden from the user of this package.  The
-- typical list manipulation operations ( e.g., as illustrated
-- by Booch 1987A and Feldman 1985 ) are only provided in the
-- package body.

-- . . .

Exception_Raiser_Record:
  Mandatory_Access_Control_Types_Package.       --  see 3 - 3.1.3
    MAC_Record_Type;          -- Initialize Exception_Raiser_Record.

Exception_Handler_Record :
  Mandatory_Access_Control_Types_Package.       --  see 3 - 3.1.3
    MAC_Record_Type;          -- Initialize Exception_Handler_Record.

-- . . .

-- Typical list manipulation operations

-- . . .

procedure Get_ACL_Record
  ( Name       : in     Name_Type;
    ACL_Record :    out ACL_Record_Type ) is

    -- . . .

    Exception_Name :
      Mandatory_Access_Control_Types_Package.    --  see 3 - 3.1.3
        Exception_Name_Type :=
          Mandatory_Access_Control_Types_Package.
            Others_String;

    -- . . .

  begin  -- Get_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- and get the indicated access control list record.
```

```
      -- . . .

    exception

      -- . . .

      when others =>
             Audit_Trail_Manager_Package.            -- see 3 - 3.2.3
               Log_Exception_in_Audit_Trail (
                 Exception_Name,
                 Exception_Raiser_Record,
                 Exception_Handler_Record );

  end Get_ACL_Record;


  procedure Insert_ACL_Record
    ( ACL_Record : in ACL_Record_Type ) is

      -- . . .

      Exception_Name :
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          Exception_Name_Type :=
            Mandatory_Access_Control_Types_Package.
              Others_String;

      -- . . .

    begin  -- Insert_ACL_Record

      -- . . .

      -- Sequence through the access control list data structure
      -- using the typical list manipulation operations to locate
      -- the appropriate place to insert the indicated
      -- access control list record.  This location is
      -- determined by a predefined mechanism, e.g., alphabetizing by
      -- the "name" of the access control list record, or more
      -- crudely by a simple (FIFO) stack or (FILO) queue.

      -- . . .

      -- Check for the exception Overflow_Access_Control_List.
      -- If the exception is to be raised, then set the
      -- Exception_Name and Exception_Raiser_Record.
    -- . . .

  exception

    -- . . .

    when Overflow_Access_Control_List =>
           Audit_Trail_Manager_Package.         -- see 3 - 3.2.3
             Log_Exception_in_Audit_Trail (
               Exception_Name,
               Exception_Raiser_Record,
               Exception_Handler_Record );

    -- . . .

    when others =>
           Audit_Trail_Manager_Package.         -- see 3 - 3.2.3
             Log_Exception_in_Audit_Trail (
               Exception_Name,
               Exception_Raiser_Record,
               Exception_Handler_Record );

end Insert_ACL_Record;

procedure Delete_ACL_Record ( Name : in Name_Type ) is

    -- . . .

    Exception_Name :
      Mandatory_Access_Control_Types_Package.   -- see 3 - 3.1.3
        Exception_Name_Type :=
          Mandatory_Access_Control_Types_Package.
            Others_String;

    -- . . .

  begin  -- Delete_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to
    -- locate, scrub, and delete the indicated
    -- access control list record.
```

```
      -- . . .

      -- Check for the exception Access_Control_List_is_Null.
      -- If the exception is to be raised, then set the
      -- Exception_Name and Exception_Raiser_Record.

      -- . . .

    exception

      -- . . .

      when Access_Control_List_is_Null =>
             Audit_Trail_Manager_Package.          --  see 3 - 3.2.3
               Log_Exception_in_Audit_Trail (
                 Exception_Name,
                 Exception_Raiser_Record,
                 Exception_Handler_Record );

      -- . . .

      when others =>
             Audit_Trail_Manager_Package.          --  see 3 - 3.2.3
               Log_Exception_in_Audit_Trail (
                 Exception_Name,
                 Exception_Raiser_Record,
                 Exception_Handler_Record );

  end Delete_ACL_Record;

  -- . . .

end Generic_Access_Control_List_Manager_Package;
```

2 - 11.1 Exception Declarations

No additional Ada-specific impact on TCBs

2 - 11.2 Exception Handlers

No additional Ada-specific impact on TCBs

2 - 11.3 Raise Statements

No additional Ada-specific impact on TCBs

2 - 11.4 Exception Handling

Exceptions in Ada are handled by the innermost execution frame or accept statement enclosing the statement that caused the exception. (Exceptions within accept statements are discussed in Section 2 - 11.5.) Because the Ada mechanism for propagating exceptions is dynamic, it deserves special attention as discussed in Section 2 - 11.

Guidelines:

1. Implement exceptions in a well-disciplined manner. Because of Ada's possible dynamic binding of exceptions to handlers, place restrictions on the implementation of exception handling to ensure that all control paths can be determined statically. At the TCB boundary, all exceptions must be handled or explicitly propagated. Similarly, implicit propagation of exceptions is disallowed within subprograms. An **others** clauses is placed in each subprogram to ensure that any exception signaled within its body will be handled. These restrictions will make control flow more predictable [ASOS 1987].

2. When a subprogram's execution is aborted because of an exception, the values **out** and **in out** parameters of array and record types is dependent on the parameter passing mechanism employed by the compiler. To foster a consistent approach, ASOS requires that the handler of an exception assume nothing about the values of the parameters [ASOS 1987].

3. Because the effects of Ada exceptions from predefined operations are not well specified, handlers shall not assume anything about the values of result variables involved in such predefined operations [ASOS 1987].

2 - 11.4.1 Exceptions Raised During the Execution of Statements

A major difficulty with exceptions in the Ada language is the dynamic manner in which exceptions are propagated and the resulting complexity that derives from attempting analysis during testing of programs. This is a specific example of the general discussion in Section 2 - 11.

2 - 11.4.2 Exceptions Raised During the Elaboration of Declarations

No additional Ada-specific impact on TCBs

2 - 11.5 Exceptions Raised During Task Communication

Exceptions raised during task communication are complicated more by the difficulty in handling tasking in Ada than by the use of exceptions in tasks.

Guideline:

1. Avoid handling an exception outside of the task that raises the exception. The use of such exceptions should be monitored during preliminary and detailed design reviews and code walkthroughs.

2 - 11.6 Exceptions and Optimization

No additional Ada-specific impact on TCBs

2 - 11.7 Suppressing Checks

Using **pragma** SUPPRESS prevents the raising of exceptions for selected checks, which can serve to monitor the proper execution of the program during runtime. Ada code generated when using **pragma** SUPPRESS cannot be trusted to work as expected because Ada compilers currently are not validated when **pragma** SUPPRESS is used.

Guideline:

1. Avoid using pragma SUPPRESS.

2 - 12. Generic Units

This chapter of the LRM describes the structure and application of generic units within Ada. The use of generic constructs is one of the more novel innovations in the Ada language. Generic units should be used in creating reusable code, as was discussed in **The Program Library** (Section 2 - 10.4).

Guideline:

1. Use generic units to write generalized software that will perform operations on classes of data types [ASOS 1987].

2 - 12.1 Generic Declarations

No Ada-specific impact on TCBs

2 - 12.1.1 Generic Format Objects

No Ada-specific impact on TCBs

2 - 12.1.2 Generic Formal Types

No Ada-specific impact on TCBs

2 - 12.1.13 Generic Formal Subprograms

No Ada-specific impact on TCBs

2 - 12.2 Generic Bodies

No Ada-specific impact on TCBs

2 - 12.3 Generic Instantiation

No Ada-specific impact on TCBs

2 - 12.3.1 Matching Rules for Formal Objects

No Ada-specific impact on TCBs

2 - 12.3.2 Matching Rules for Formal Private Types

No Ada-specific impact on TCBs

2 - 12.3.3 Matching Rules for Formal Scalar Types

No Ada-specific impact on TCBs

2 - 12.3.4 Matching Rules for Formal Array Types

No Ada-specific impact on TCBs

2 - 12.3.5 Matching Rules for Formal Access Types

No Ada-specific impact on TCBs

2 - 12.3.6 Matching Rules for Formal Subprograms

2 - 12.4 Example of a Generic Package

Example: This example illustrates a generic package for multiple instances of an access control list, with instantiations of the package.

```
generic

  -- . . .

  type Name_Type is private;
  type ACL_Record_Type is private;

  -- . . .

package Generic_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in     Name_Type;
        ACL_Record :    out ACL_Record_Type );
    procedure Insert_ACL_Record
      ( ACL_Record : in ACL_Record_Type );

    procedure Delete_ACL_Record
      ( Name : in Name_Type );

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;

  private

    -- . . .

end Generic_Access_Control_List_Manager_Package;


with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;       -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;     -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                  -- see 3 - 3.2.3
package body Generic_Access_Control_List_Manager_Package is

    -- The user can gain only indirect access to instantiated
    -- access control list through the subprograms declared in the
    -- package specification.  Thus the access control list data
    -- structure is hidden from the user of this package.  The
    -- typical list manipulation operations ( e.g., as illustrated
    -- by Booch 1987A and Feldman 1985 ) are only provided in the
    -- package body.

    -- . . .

    -- Typical list manipulation operations

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in     Name_Type;
        ACL_Record :    out ACL_Record_Type ) is

        -- . . .
  begin  -- Get_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- and get the indicated access control list record.

    -- . . .

end Get_ACL_Record;


procedure Insert_ACL_Record
  ( ACL_Record : in ACL_Record_Type ) is

    -- . . .

  begin  -- Insert_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- the appropriate place to insert the indicated
    -- access control list record.  This location is
    -- determined by a predefined mechanism, e.g., alphabetizing by
    -- the "name" of the access control list record, or more
    -- crudely by a simple (FIFO) stack or (FILO) queue.

    -- . . .

end Insert_ACL_Record;


procedure Delete_ACL_Record ( Name : in Name_Type ) is

    -- . . .

  begin  -- Delete_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate,
    -- scrub, and delete the indicated access control list record.
```

```
          -- . . .

      end Delete_ACL_Record;

      -- . . .

  end Generic_Access_Control_List_Manager_Package;


  -- Instantiations of Generic_Access_Control_List_Manager_Package
  package Named_Objects_Access_Control_List_Manager_Package is new
          Generic_Access_Control_List_Manager_Package
            ( Name_Type       => Basic_TCB_Types_Package.
                                   Name_of_Object_Type,
              ACL_Record_Type => Access_Control_List_Types_Package.
                                   Named_Object_Record_Type );


  package Named_Individuals_Access_Control_List_Manager_Package is new
          Generic_Access_Control_List_Manager_Package
            ( Name_Type       => Basic_TCB_Types_Package.
                                   Name_of_Object_Type,
              ACL_Record_Type => Access_Control_List_Types_Package.
                                   Named_Individual_Record_Type );


  package Groups_of_Named_Individuals_ACL_Manager_Package is new
          Generic_Access_Control_List_Manager_Package
            ( Name_Type       => Basic_TCB_Types_Package.
                                   Name_of_Object_Type,
              ACL_Record_Type => Access_Control_List_Types_Package.
                                 Group_of_Named_Individuals_Record_Type );
```

2 - 13. Representation Clauses and Implementation-Dependent Features

This chapter of the LRM discusses implementation-specific matters at a low level. Several of the constructs, such as representation clauses, length clauses, enumeration representation clauses, and address clauses are on the order of specific directives to the compiler and would have no noticeable effect on the execution of the resulting program. Interrupts are discussed in Section 2 - 13.5.1. Machine code insertions (Section 2 - 13.8) and interfaces to subprograms written in other languages (Section 2 - 13.9) may introduce complications in developing TCBs. Also of interest to the development TCBs is unchecked programming (Section 2 - 13.10).

Guidelines:

1. The use of representation clauses and implementation-dependent features should be avoided in Ada TCB system implementations.

2. Security aspects of using any representation clauses and implementation-dependent features should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

3. The use of any representation clauses and implementation-dependent features should be restricted to a minimum number of packages and subprograms to assist in monitoring their use and to aid in system portability.

2 - 13.1 Representation Clauses

No additional Ada-specific impact on TCBs

2 - 13.2 Length Clauses

No additional Ada-specific impact on TCBs

2 - 13.3 Enumeration Representation Clauses

No additional Ada-specific impact on TCBs

2 - 13.4 Record Representation Clauses

No additional Ada-specific impact on TCBs

2 - 13.5 Address Clauses

No additional Ada-specific impact on TCBs

2 - 13.5.1 Interrupts

Interrupts provide asynchronous means of altering program execution, so that an external event can be handled by the system. This change in program execution must be managed and documented.

Guidelines:

1. Each interrupt should be labeled so that the initiator of the interrupt is known by its interrupt handler. Interrupts should be managed with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). Interrupts should be monitored by logging them in the audit trail.

2. All interrupts that affect a TCB must be raised within the TCB and handled by the TCB.

3. Security aspects of the various interrupt operations should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications.

4. Implementation of these interrupt handling operations should be monitored during the TCB system's preliminary and detailed design reviews and code walkthroughs.

Example: This example illustrates trusted interrupt handling.
Note that address clauses are not currently supported in VAX Ada.

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
package Interrupt_Handler_Package is

  procedure Get_Character ( Char              :   out CHARACTER;
                            Local_MAC_Record : in
                              Mandatory_Access_Control_Types_Package.
                                MAC_Record_Type );  -- see 3 - 3.1.3
  procedure Put_Character ( Char             : in CHARACTER;
                            Local_MAC_Record : in
                              Mandatory_Access_Control_Types_Package.
                                MAC_Record_Type ); -- see 3 - 3.1.3

end Interrupt_Handler_Package;


with Mandatory_Access_Control_Manager_Package;       -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    -- see 3 - 3.2.3
package body Interrupt_Handler_Package is

  -- . . .

  task Interrupt_Input_Handler_Task is

    pragma PRIORITY ( 4 );
         -- must have at least the priority of the interrupt

    entry Get_Character_from_Interrupt_Input_Address
            ( Char                     : out CHARACTER;
              Other_Task_MAC_Record : in
                Mandatory_Access_Control_Types_Package.
                  MAC_Record_Type );                 -- see 3 - 3.1.3

    entry Save_Hardware_Buffer_Character (
            Other_Task_MAC_Record : in
              Mandatory_Access_Control_Types_Package.
                MAC_Record_Type );                   -- see 3 - 3.1.3

    -- assuming that SYSTEM. ADDRESS is an INTEGER type
    for Save_Hardware_Buffer_Character use at 16#0020#;

  end Interrupt_Input_Handler_Task;


  task Interrupt_Output_Handler_Task is

    pragma PRIORITY ( 4 );
         -- must have at least the priority of the interrupt

    entry Deposit_Character_into_Hardware_Buffer (
            Other_Task_MAC_Record : in
              Mandatory_Access_Control_Types_Package.
                MAC_Record_Type );                   -- see 3 - 3.1.3
  entry Put_Character_into_Interrupt_Output_Address (
          Char                    : in CHARACTER;
          Other_Task_MAC_Record : in
            Mandatory_Access_Control_Types_Package.
              MAC_Record_Type );                    -- see 3 - 3.1.3

  -- assuming that SYSTEM. ADDRESS is an INTEGER type
  for Deposit_Character_into_Hardware_Buffer use at 16#0024#;

end Interrupt_Output_Handler_Task;


procedure Get_Character ( Char             :    out CHARACTER;
                          Local_MAC_Record : in
                            Mandatory_Access_Control_Types_Package.
                              MAC_Record_Type ) is
                                                    -- see 3 - 3.1.3
  begin  -- Get_Character

    Interrupt_Input_Handler_Task.
      Get_Character_from_Interrupt_Input_Address
        ( Char, Local_MAC_Record  );

end Get_Character;


procedure Put_Character ( Char             : in CHARACTER;
                          Local_MAC_Record : in
                            Mandatory_Access_Control_Types_Package.
                              MAC_Record_Type ) is
                                                    -- see 3 - 3.1.3
  begin  -- Put_Character

    Interrupt_Output_Handler_Task.
      Put_Character_into_Interrupt_Output_Address
        ( Char, Local_MAC_Record );

end Put_Character;


task body Interrupt_Input_Handler_Task is

    Max_Size_of_Internal_Input_Buffer : constant POSITIVE
      := 64;
      Internal_Input_Buffer :
        array ( 1 .. Max_Size_of_Internal_Input_Buffer )
          of CHARACTER;

      Input_Buffer_Pointer  : POSITIVE := 1;
      Output_Buffer_Pointer : POSITIVE := 1;

      Buffer_Count : INTEGER := 1;

      Hardware_Character_Buffer : CHARACTER;
      for Hardware_Character_Buffer use at 16#0100#;

      Local_Input_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
          MAC_Record_Type;

      Other_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
          MAC_Record_Type;

    begin  -- Interrupt_Input_Handler_Task
      loop
        select
            when Buffer_Count > 0 =>

              accept Get_Character_from_Interrupt_Input_Address
                       ( Char                  :    out CHARACTER;
                         Other_Task_MAC_Record : in
                           Mandatory_Access_Control_Types_Package.
                             MAC_Record_Type ) do
                                                     --  see 3 - 3.1.3

                if Mandatory_Access_Control_Manager_Package.
                     Sensitivity_Labels_Match        --  see 3 - 3.1.3
                       ( Local_Input_Task_MAC_Record.
                           Sensitivity_Label,
                         Other_Task_MAC_Record.
                           Sensitivity_Label ) then

                  Audit_Trail_Manager_Package.       --  see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_Input_Task_MAC_Record,
                      Other_Task_MAC_Record );

                  Char := Internal_Input_Buffer(
                            Output_Buffer_Pointer );
                  Output_Buffer_Pointer :=
                    Output_Buffer_Pointer mod
                      Max_Size_of_Internal_Input_Buffer + 1;

                  Buffer_Count := Buffer_Count - 1;
                end if;
              end Get_Character_from_Interrupt_Input_Address;

          or
            when Buffer_Count <
                   Max_Size_of_Internal_Input_Buffer =>

              accept Save_Hardware_Buffer_Character (
                       Other_Task_MAC_Record : in
                         Mandatory_Access_Control_Types_Package.
                           MAC_Record_Type ) do      --  see 3 - 3.1.3

                if Mandatory_Access_Control_Manager_Package.
                     Sensitivity_Labels_Match          --  see 3 - 3.1.3
                       ( Local_Input_Task_MAC_Record.
                           Sensitivity_Label,
                         Other_Task_MAC_Record.
                           Sensitivity_Label ) then

                  Audit_Trail_Manager_Package.         --  see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_Input_Task_MAC_Record,
                      Other_Task_MAC_Record );

                  Internal_Input_Buffer(
                    Input_Buffer_Pointer ) :=
                      Hardware_Character_Buffer;

                  Input_Buffer_Pointer :=
                    Input_Buffer_Pointer mod
                      Max_Size_of_Internal_Input_Buffer + 1;

                  Buffer_Count := Buffer_Count + 1;
                end if;
              end Save_Hardware_Buffer_Character;
          end select;
        end loop;
    end Interrupt_Input_Handler_Task;
```

```
task body Interrupt_Output_Handler_Task is

    Max_Size_of_Internal_Output_Buffer : constant POSITIVE
      := 64;

    Internal_Output_Buffer :
      array ( 1 .. Max_Size_of_Internal_Output_Buffer )
        of CHARACTER;

    Input_Buffer_Pointer  : POSITIVE := 1;
    Output_Buffer_Pointer : POSITIVE := 1;

    Buffer_Count : INTEGER := 1;

    Hardware_Character_Buffer : CHARACTER;
    for Hardware_Character_Buffer use at 16#0200#;

    Hardware_Character_Buffer_is_Empty : BOOLEAN := TRUE;

    Local_Output_Task_MAC_Record :
      Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
        MAC_Record_Type;

    Other_Task_MAC_Record :
      Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
        MAC_Record_Type;

  begin  -- Interrupt_Output_Handler_Task
    loop
      select
          accept Deposit_Character_into_Hardware_Buffer (
                   Other_Task_MAC_Record : in
                     Mandatory_Access_Control_Types_Package.
                       MAC_Record_Type ) do        -- see 3 - 3.1.3

              if Mandatory_Access_Control_Manager_Package.
                   Sensitivity_Labels_Match        -- see 3 - 3.1.3
                     ( Local_Output_Task_MAC_Record.
                         Sensitivity_Label,
                       Other_Task_MAC_Record.
                         Sensitivity_Label ) then

                Audit_Trail_Manager_Package.       -- see 3 - 3.2.3
                  Log_Subjects_Access_to_Object_in_Audit_Trail (
                    Local_Output_Task_MAC_Record,
                    Other_Task_MAC_Record );
          if Buffer_Count > 0 then

            Hardware_Character_Buffer :=
              Internal_Output_Buffer(
                Output_Buffer_Pointer );

            Output_Buffer_Pointer :=
              Output_Buffer_Pointer mod
                Max_Size_of_Internal_Output_Buffer + 1;

            Buffer_Count := Buffer_Count - 1;

          else

            Hardware_Character_Buffer_is_Empty := TRUE;
        end if;
      end if;
    end Deposit_Character_into_Hardware_Buffer;

  or
    when Buffer_Count <
           Max_Size_of_Internal_Output_Buffer =>

      accept Put_Character_into_Interrupt_Output_Address
              ( Char                    : in CHARACTER;
                Other_Task_MAC_Record : in
                  Mandatory_Access_Control_Types_Package.
                    MAC_Record_Type ) do    --  see 3 - 3.1.3

        if Mandatory_Access_Control_Manager_Package.
             Sensitivity_Labels_Match          --  see 3 - 3.1.3
               ( Local_Output_Task_MAC_Record.
                   Sensitivity_Label,
                 Other_Task_MAC_Record.
                   Sensitivity_Label ) then

          Audit_Trail_Manager_Package.        --  see 3 - 3.2.3
            Log_Subjects_Access_to_Object_in_Audit_Trail (
              Local_Output_Task_MAC_Record,
              Other_Task_MAC_Record );

          Internal_Output_Buffer(
            Input_Buffer_Pointer ) := Char;
```

```
                  Input_Buffer_Pointer :=
                    Input_Buffer_Pointer mod
                      Max_Size_of_Internal_Output_Buffer + 1;

                  Buffer_Count := Buffer_Count + 1;

                  if Hardware_Character_Buffer_is_Empty then

                    Hardware_Character_Buffer :=
                      Internal_Output_Buffer(
                        Output_Buffer_Pointer );

                    Output_Buffer_Pointer :=
                      Output_Buffer_Pointer mod
                        Max_Size_of_Internal_Output_Buffer + 1;

                    Buffer_Count := Buffer_Count - 1;

                    Hardware_Character_Buffer_is_Empty := TRUE;
                  end if;
                end if;
              end Put_Character_into_Interrupt_Output_Address;
        end select;
      end loop;
  end Interrupt_Output_Handler_Task;

end Interrupt_Handler_Package;
```

2 - 13.6 Change of Representation

The change of representation clauses introduces potential problems similar to those associated with type conversion (Section 2 - 4.6).

Guideline:
1. The change of any representation clause should be monitored during design reviews and code walkthroughs.

2 - 13.7 The Package System

No Ada-specific impact on TCBs

2 - 13.7.1 System-Dependent Named Numbers

No Ada-specific impact on TCBs

2 - 13.7.2 Representation Attributes

No Ada-specific impact on TCBs

2 - 13.7.3 Representation Attributes of Real Types

No Ada-specific impact on TCBs

2 - 13.8 Machine Code Insertions

One use of this feature is to insert calls to currently existing functions (e.g., sort routines). A specification of the routines at the Ada specification language level and establishing that the routines do not stray from their specified behavior are necessary prerequisites to use of machine code insertions.

Guideline:
1. New software components that have traditionally been written in machine code (e.g., device drivers which handle sensitivity labels in input or output operations) should be written in Ada. If "trusted" machine code already exists, then it may be acceptable to use it rather than writing new Ada code.

2 - 13.9 Interface to Other Languages

Ada provides the means to interface with other languages using the pragma INTERFACE. Using multiple high-order languages, or using assembly language with a high-order language, makes a system more difficult to understand. The difficulty here is not with the pragma INTERFACE per se, but rather with using more than one language in a system.

Guideline:
1. Avoid interfacing Ada with another language.

Example: This example illustrates the interface of Ada with Pascal.

```
package Graphics_Library_Package is
    procedure Draw_Circle
                ( Center : in    Coordinates_Type;
                  Radius : in    Distance_Type );

    -- . . .

  private
    pragma INTERFACE ( PASCAL, Draw_Circle );

    -- . . .

end Graphics_Library_Package;
```

2 - 13.10 Unchecked Programming

Ada provides two unchecked programming features, unchecked storage deallocation and unchecked type conversion. "The predefined generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are used for unchecked storage deallocation and for unchecked type conversions" [ANSI/MIL-STD-1815A-1983].

2 - 13.10.1 Unchecked Storage Deallocation

The only program-visible effect of using unchecked_deallocation is the assignment of the access value null to the variable being deallocated. This does not ensure that the variable is scrubbed before it is deallocated.

Guideline:
1. Avoid using the unchecked programming feature UNCHECKED_DEALLOCATION.

2 - 13.10.2 Unchecked Type Conversions

One major difficulty with the use of unchecked type conversions is specifying the transformation between the two types that takes place during the conversion.

Guideline:
1. Avoid using the unchecked programming feature UNCHECKED_CONVERSION.

2 - 14. Input-Output

This chapter of the LRM describes the mechanisms for input and output for an Ada program and the management of file objects. The packages described include procedures for the input of sequential, direct, numeric data, and enumeration data. A general discussion of input and output operations is located in this section.

The major obstacles to input and output are the lack of the semantics of input and output, and the ability to do input and output anywhere within an Ada program.

An example for handling input and output is provided in Section 2 - 14.7.

Guidelines:

1. Minimize the use of input and output [ASOS 1987].

2. Security aspects of all input and output operations, such as maintaining the secrecy of the information between the TCB and displays, disks, and tapes, should be addressed as early as possible in the development of a TCB system, preferably in its requirements specification.

3. The security of the TCB's input and output functions should be provided with discretionary access controls (e.g., access control lists) and/or mandatory access controls (e.g., sensitivity labels). All inputs to and outputs from a TCB should be logged in the audit trail.

4. Security aspects of input and output operations should be addressed as early as possible in the development of a TCB system, preferably in its requirement specifications. These operations should be conscientiously implemented as stipulated by the TCB system's requirements specification, preliminary and detailed design reviews and code walkthroughs.

2 - 14.1 External Files and File Objects

No additional Ada-specific impact on TCBs

2 - 14.2 Sequential and Direct Files

No additional Ada-specific impact on TCBs

2 - 14.2.1 File Management

No additional Ada-specific impact on TCBs

2 - 14.2.2 Sequential Input-Output

No additional Ada-specific impact on TCBs

2 - 14.2.3 Specification of the Package Sequential_IO

No additional Ada-specific impact on TCBs

2 - 14.2.4 Direct Input-Output

No additional Ada-specific impact on TCBs

2 - 14.2.5 Specification of the Package Direct_IO

No additional Ada-specific impact on TCBs

2 - 14.3 Text Input-Output

No additional Ada-specific impact on TCBs

2 - 14.3.1 File Management

No additional Ada-specific impact on TCBs

2 - 14.3.2 Default Input and Output Files

No additional Ada-specific impact on TCBs

2 - 14.3.3 Specification of Line and Page Lengths

No additional Ada-specific impact on TCBs

2 - 14.3.4 Operations on Columns, Lines, and Pages

No additional Ada-specific impact on TCBs

2 - 14.3.5 Get and Put Procedures

No additional Ada-specific impact on TCBs

2 - 14.3.6 Input-Output of Characters and Strings

No additional Ada-specific impact on TCBs

2 - 14.3.7 Input-Output for Integer Types

No additional Ada-specific impact on TCBs

2 - 14.3.8 Input-Output for Real Types

No additional Ada-specific impact on TCBs

2 - 14.3.9 Input-Output for Enumeration Types

No additional Ada-specific impact on TCBs

2 - 14.3.10 Specification of the Package Text_IO

No additional Ada-specific impact on TCBs

2 - 14.4 Exceptions in Input-Output

No additional Ada-specific impact on TCBs

2 - 14.5 Specification of the Package IO_Exceptions

No additional Ada-specific impact on TCBs

2 - 14.6 Low Level Input-Output

No additional Ada-specific impact on TCBs

2 - 14.7 Example of Input-Output

Example: This example illustrates trusted input and output.

```
with TEXT_IO;
with Mandatory_Access_Control_Types_Package;         --  see 3 - 3.1.3
with Access_Control_List_Types_Package;
generic

    Max_Characters_in_Message : POSITIVE := 80;

package Generic_Sensitive_Text_File_Manager_Package is

  Sensitive_Text_File : TEXT_IO. FILE_TYPE;
  subtype Message_Type is STRING( 1 .. Max_Characters_in_Message );

  -- . . .

  procedure Put_Line (
    Subject_MAC_Record       : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Named_Object_MAC_Record : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Message                  : in Message_Type );

  -- . . .

end Generic_Sensitive_Text_File_Manager_Package;


with Mandatory_Access_Control_Manager_Package;       --  see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    --  see 3 - 3.2.3
package Generic_Sensitive_Text_File_Manager_Package is

  -- . . .
```

```
procedure Put_Line (
    Subject_MAC_Record       : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Named_Object_MAC_Record : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Message                  : in Message_Type ) is

  begin  -- Put_Line

    if Mandatory_Access_Control_Manager_Package.
         Sensitivity_Labels_Match                    -- see 3 - 3.1.3
           ( Subject_MAC_Record. Sensitivity_Label,
             Named_Object_MAC_Record. Sensitivity_Label ) then

      Audit_Trail_Manager_Package.                   -- see 3 - 3.2.3
        Log_Subjects_Access_to_Object_in_Audit_Trail (
          Subject_MAC_Record, Named_Object_MAC_Record );

      TEXT_IO. PUT_LINE ( Sensitive_Text_File, Message );

    end if;
end Put_Line;

-- . . .

end Generic_Sensitive_Text_File_Manager_Package;
```

2.1 Tailoring and Configuring Ada Compiler and Runtime System

This section does not have a corresponding chapter in the LRM. Nevertheless, it is an important issue that should be addressed in the software development of TCB systems.

Tailoring an Ada compiler or runtime system is the actual modification of the code of the Ada compilation system [Baker 1988]. Configuration of an Ada compiler or runtime system is the reselection of compiler options and parameters provided by the Ada compiler vendor. Tailoring and configuration may allow a compiler or runtime system to run conveniently on various host and target combinations. The compiler will no longer be validated, however, and other problems may be introduced during the tailoring or configuring process. Modifying the Ada compiler or runtime system of a TCB by tailoring or configuring should be avoided.

Guidelines:

1. Any tailoring or configuring of the Ada compiler or runtime system must be done under the supervision of and with the approval of the TCB system security administrator.

2. Security aspects of any tailoring of an Ada compiler or runtime system should be addressed as early as possible in the development of a TCB system, preferably in its requirements specification.

3. Any element of an Ada runtime environment that is tailored should be subjected to the same evaluation criteria that are applied to the TCB being developed.

3.0 MAPPING OF ADA USAGE TO TCB CRITERIA

This section provides a mapping of Ada constructs that would be appropriate to the implementation of the TCB structures and functions defined in the TCSEC. The class B3 criteria are used as the template of the generalized TCB criteria, because they are the highest level of security criteria under consideration. Also, the Ada constructs mapped to these criteria can similarly be mapped to the TCB criteria of lower security classes. The four TCB criteria subsections considered are Security Policy, Accountability, Assurance, and Documentation. For further discussion of the various topics, refer to the TCSEC.

GENERALIZED TCB CRITERIA

"The class B3 TCB must satisfy the reference monitor requirements that it mediate all accesses of objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. A security administrator is supported, audit mechanisms are expanded to signal security-relevant events, and system recovery procedures are required. The system is highly resistant to penetration."

3.1 Security Policy

A security policy describes how users may access documents or other information. It is the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

3.1.1 Discretionary Access Control

Discretionary access control provides a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) to any other subject (unless restrained by mandatory access control). An enforcement mechanism (e.g., access control lists) must allow users to specify and control sharing of those objects and must provide controls to limit propagation of access rights.

Guidelines for the use of Ada Constructs and Features:

Enforcement mechanisms that consist of lists, such as access control lists, should be created with linked lists and queues, using either static or dynamic storage constructs (e.g., using arrays or access types, respectively). Though linked lists and queues are more typically implemented using dynamic storage constructs created with Ada's access types, which provide more efficient memory usage, linked lists and queues created with arrays provide more efficient execution. In addition, the lists should be represented by abstract data types, which consist of Ada's strong data typing encapsulated in packages. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
ARRAYS	2 - 3.6
PACKAGES	2 - 7.
TYPES	2 - 3.

Example: This example illustrates a generic package specification for multiple instances of user identification and authentication (The package body is in Section 3 - 3.2.1.)

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
with Generic_Access_Control_List_Manager_Package;
generic

  -- . . .

  type Password_Type is private;

  -- . . .

package Generic_User_Identification_and_Authentication_Package is

    -- . . .

    procedure Check_Password
      ( Password            : in     Password_Type;

        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
            MAC_Record_Type;

        Password_is_Valid :   out BOOLEAN );

    -- . . .
    end Generic_User_Identification_and_Authentication_Package;
```

Use tasking to handle concurrent processes in more complex TCB systems where multiuser and multisubject requirements (e.g., simultaneously monitoring accesses to control lists by multiple subjects) are present. Tasks are discussed in Section 2 - 9.

When nonvolatile versions of major lists (e.g., access lists, need to be accessed from disk or tape) protect the security of input and output operations with protocols that satisfy the class of the given TCB. Input and output is discussed in Section 2 - 14.

3.1.2 Object Reuse

Object reuse is the reassignment to some subject of a medium (e.g., page frame disk sector, magnetic tape) that contained one or more objects. To be securely reassigned, such media must contain no residual data from the previously contained object(s). The TCB must assure that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the object contains no data for which the subject is not authorized.

Guidelines for the use of Ada Constructs and Features:

The reuse of objects involves the management of memory used for storing objects. Dynamic storage as well as static storage should be managed in a secure manner. Also, objects should be represented by abstract data types, which are implemented with Ada's packages and user-defined data types. Use tasking and shared variables as required to manage object reuse when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
ARRAYS	2 - 3.6
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

3.1.3 Labels

A sensitivity label is a piece of information that represents the security level of an object and that describes the sensitivity (i.e., classification) of the data in the object. Sensitivity labels are used by the TCB as the basis for mandatory access control decisions. They are associated with each ADP system resource (e.g., subject, storage object) that is directly or indirectly accessible by subjects external to the TCB, and they must be maintained by the TCB. To import non-labeled data, the TCB must request and receive from an authorized user the security level of the data, and all such actions must be auditable by the TCB. Also, the TCB must enforce subject sensitivity labels and device labels.

Guidelines for the use of Ada Constructs and Features:

Treat sensitivity labels as objects that are represented by abstract data types. These should consist of Ada's strong data typing encapsulated in packages. The exportation of labeled information typically involves input and output using secure protocols, which should also be represented by abstract data types. Use tasking and shared variables as required to manage subject sensitivity labels and their exportation when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

INPUT and OUTPUT	2 - 14.
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

Example: This example illustrates sensitivity label type declaration.

```
with Basic_TCB_Types_Package;
package Mandatory_Access_Control_Types_Package is

  -- . . .

  subtype Name_of_Resource_Type is
          Basic_TCB_Types_Package. Name_String_Type;

  subtype Sensitivity_Label_Type is
          Basic_TCB_Types_Package. Name_String_Type;

  Scrubbed_Sensitivity_Label : Sensitivity_Label_Type
    := Basic_TCB_Types_Package. Blank_Name_String;

  subtype Exception_Name_Type is
          Basic_TCB_Types_Package. Name_String_Type;
```

```
   Scrubbed_Exception_Name : Exception_Name_Type
     := Basic_TCB_Types_Package. Blank_Name_String;

   Others_String : Basic_TCB_Types_Package. Name_String_Type
     := Basic_TCB_Types_Package. Blank_Name_String;

   type MAC_Record_Type is
     record
       Name · Name_of_Resource_Type
         := Basic_TCB_Types_Package. Blank_Name_String; -- see 2 - 3.2.1

       Sensitivity_Label : Sensitivity_Label_Type
         := Scrubbed_Sensitivity_Label;

       -- . . .

     end record;

   -- . . .

 end Mandatory_Access_Control_Types_Package;


 with Basic_TCB_Types_Package;
 with Mandatory_Access_Control_Types_Package;
 package Mandatory_Access_Control_Manager_Package is

   -- . . .

   function Sensitivity_Labels_Match
     ( Sensitivity_Label_A : in
         Mandatory_Access_Control_Types_Package.
           Sensitivity_Label_Type;

       Sensitivity_Label_B : in
         Mandatory_Access_Control_Types_Package.
           Sensitivity_Label_Type )

     return BOOLEAN;

   -- . . .

 end Mandatory_Access_Control_Manager_Package;
```

3.1.4 Mandatory Access Control

A mandatory access control is a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. It prevents "some types of Trojan horse attacks by imposing access restrictions that cannot be bypassed, even indirectly. Under mandatory controls, the system assigns both subjects and objects special attributes that cannot be changed on request as can discretionary access control attributes such as access control lists. The system decides whether a subject can access an object by comparing their security attributes" [Gasser 1988, p.61]. Thus, a TCB must enforce a mandatory access control policy over all resources (i.e., subjects, storage objects and I/O devices) that are directly or indirectly accessible by subjects external to the TCB. All subjects and storage objects must be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels must be the basis of the mandatory access control decisions. Also, a TCB must be able to support two or more security levels.

Guidelines for the use of Ada Constructs and Features:

The management of a mandatory access control policy is performed typically with an implementation of the Bell and LaPadula security model that regulates the security of the accessing of objects by subjects and the assignment of sensitivity labels to enforce the policy. This requires classifications of the objects which are to be protected by this policy. Manage dynamic storage as well as static storage in a secure manner. Also, objects should be represented by abstract data types, which are implemented with Ada's packages and user-defined data types. Similarly, the policy itself should be represented by a package. Use tasking and shared variables as required to manage mandatory access controls when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
ARRAYS	2 - 3.6
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

3.2 Accountability

Accountability is the monitoring of access to and operation of a TCB system by using identification and authentication of users requesting access to the system, maintenance of trusted communication paths, and auditing of accesses to the TCB.

3.2.1 Identification and Authentication

Identification consists of using unique identifiers that are associated with each user (such as a last name, initials, or account number) that everyone knows, that nobody can forge or change, and that all access requests can be checked against. The identifier is the means by which the system distinguishes users. In particular, a TCB must require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate.

Authentication consists of associating a real user (or more accurately, a program running on behalf of a user) with a unique identifier, namely, the user ID. "The system must separate authentication information (passwords) from identification information (unique IDs) to the maximum extent possible, because passwords are secret and user IDs are public" [Gasser 1988, p.23]. A TCB must maintain authentication data that includes information for verifying the identity of individual users as well as information for determining the clearance and authorizations of individual users. It uses this data to authenticate the user's identity and to determine the security level and authorizations of subjects that are created to act on the behalf of the individual user. The TCB must protect authentication data so that it cannot be accessed by an unauthorized user. It must be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. Also, it must provide the capability of associating the individual identity with all auditable actions taken by that individual.

Guidelines for the use of Ada Constructs and Features:

Manage identification and authentication data using corresponding abstract data types, which consist of Ada's strong data typing encapsulated in packages. This management should also include using dynamic storage as well as static storage in a secure manner. Similarly, the identification and authentication processes should be represented by packages. Use tasking and shared variables as required to manage security data when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
ARRAYS	2 - 3.6
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

Example: This example illustrates a generic package body for multiple instances of user identification and authentication. (The package specification is in Section 3 - 3.1.1.)

```
with Access_Control_List_Types_Package;
with Audit_Trail_Manager_Package;                       -- see 3 - 3.2.3
package body Generic_User_Identification_and_Authentication_Package is

    -- . . .

    procedure Check_Password
      ( Password          : in     Password_Type;

        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.       -- see 3 - 3.1.3
            MAC_Record_Type;

        Password_is_Valid :   out BOOLEAN ) is

        -- . . .

      begin  -- Check_Password

        -- . . .

    end Check_Password;

    -- . . .

end Generic_User_Identification_and_Authentication_Package;
```

3.2.2 Trusted Path

A trusted path is a mechanism by which a person at a terminal can communicate directly with the TCB. This mechanism can only be activated by the person or the TCB and cannot be imitated by unevaluated software. A TCB must support a trusted communication path between itself and users for use when a positive TCB-to-user connection is required (e.g., login, change subject security level). Communication via this trusted path must be activated either by a user or the TCB and must be logically isolated and unmistakably distinguishable from other paths. The Trusted Path for a Class B3 TCB needs to allow bi-directional access, from user to TCB and from TCB to user.

Guidelines for the use of Ada Constructs and Features:

Trusted paths require secure input and output communication paths between the user and subject and the TCB. Trusted paths should be treated as objects, which can be represented by abstract data types. These should consist of Ada's strong data typing encapsulated in packages. Use tasking and shared variables as required to manage trusted paths when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

INPUT and OUTPUT	2 - 14
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

3.2.3 Audit

An audit of accesses to and operation of TCB operation is maintained in a set of records (i.e., an audit trail) that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions. The TCB must be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects. Audit data must be protected by the TCB so that read access to it is limited to those who are authorized for audit data. A TCB must be able to record use of identification and authentication mechanisms, introduction of objects into a user's address space, deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events. A TCB must be able to audit any override of human-readable output markings. For each recorded event, the audit record must identify date and time of event, user, type of event, and success or failure of the event. For identification and authentication events, the audit record must include origin of request (terminal ID). For events that introduce an object into a user's address space and for object deletion events, the audit record must include the name of the object and the object's security level. The ADP system administrator must be able to selectively audit the actions of any one or more users based on individual identity and/or object security level. A TCB must be able to audit the identified events that may be used in the exploitation of covert storage channels. A TCB must contain a mechanism that is able to monitor the occurrence or accumulation of security auditable events that may indicate an imminent violation of security policy; this mechanism must be able to immediately notify the security administrator when thresholds are exceeded. If the occurrence or accumulation of these security relevant events continues, the system must take the least disruptive action to terminate the event.

Guidelines for the use of Ada Constructs and Features:

Managing audit data requires maintaining an audit trail. An audit trail is typically recorded on disk and/or tape, which requires secure input and output communication paths within a TCB that includes a secure disk and/or tape. The audit data and audit trail should be treated as objects, which can be represented by abstract data types. These should consist of Ada's strong data typing encapsulated in packages. In addition to being stored on disk or tape, the audit data should be (at least partially) located in dynamic storage or static storage. Use tasking and shared variables as required to manage the audit data and audit trail when concurrent processes are warranted for efficient multiuser and multisubject TCB system operation. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
ARRAYS	2 - 3.6
INPUT and OUTPUT	2 - 14
PACKAGES	2 - 7.
TASKS	2 - 9.
TYPES	2 - 3.

Example: Audit trail manager package

```
with TEXT_IO;
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
package Audit_Trail_Manager_Package is

  -- . . .

  Audit_Trail_Text_File : TEXT_IO. FILE_TYPE;

  -- . . .

  procedure Log_Subjects_Access_to_Object_in_Audit_Trail (
    Subject_MAC_Record      : in
      Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
        MAC_Record_Type;

    Named_Object_MAC_Record : in
      Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
        MAC_Record_Type );

  -- . . .
  procedure Log_Exception_in_Audit_Trail (
    Exception_Name            : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        Exception_Name_Type;

    Exception_Raiser_Record   : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type;

    Exception_Handler_Record : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type );

  -- . . .

end Audit_Trail_Manager_Package;


with Mandatory_Access_Control_Manager_Package;
package body Audit_Trail_Manager_Package is

    -- . . .

    procedure Log_Subjects_Access_to_Object_in_Audit_Trail (
      Subject_MAC_Record        : in
        Mandatory_Access_Control_Types_Package.       -- see 3 - 3.1.3
          MAC_Record_Type;

      Named_Object_MAC_Record : in
        Mandatory_Access_Control_Types_Package.       -- see 3 - 3.1.3
          MAC_Record_Type ) is

        -- . . .

      begin  -- Log_Subjects_Access_to_Object_in_Audit_Trail

        -- . . .

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "Subject " & Subject_MAC_Record. Name );

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "  is not authorized to access " &
                             Named_Object_MAC_Record. Name );
        -- . . .

    end Log_Subjects_Access_to_Object_in_Audit_Trail;
```

```
-- . . .

procedure Log_Exception_in_Audit_Trail (
  Exception_Name           : in
    Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
      Exception_Name_Type;

  Exception_Raiser_Record  : in
    Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
      MAC_Record_Type;

  Exception_Handler_Record : in
    Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
      MAC_Record_Type ) is

    -- . . .

  begin  -- Log_Exception_in_Audit_Trail

    -- . . .

    if Mandatory_Access_Control_Manager_Package.
         Sensitivity_Labels_Match                -- see 3 - 3.1.3
           ( Exception_Handler_Record.
               Sensitivity_Label,
              Exception_Raiser_Record.
                Sensitivity_Label ) then

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "Subprogram " &
                            Exception_Handler_Record. Name &
                            " handled the exception, " );

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File, "  ", &
                            Exception_Name & " raised by " &
                            Exception_Raiser_Record. Name );

      else

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "Subprogram " &
                            Exception_Handler_Record. Name &
                   " is not authorized to handle the exception, " );
```

```
            TEXT_IO. PUT_LINE ( Audit_Trail_Text_File, "  ", &
                                Exception_Name & " raised by " &
                                Exception_Raiser_Record. Name );
        end if;

        -- . . .

      end Log_Exception_in_Audit_Trail;

      -- . . .

    begin  -- Audit_Trail_Manager_Package initialization

      Mandatory_Access_Control_Types_Package.          -- see 3 - 3.1.3
        Others_String( 1 .. 6 ) := "others";

  end Audit_Trail_Manager_Package;
```

3.3 Assurance

Assurance of the correctness of a TCB system's security controls, as specified by the security requirements, determine the extent that the security architecture must dictate many details of the development process. The two types of assurance that must be considered are operational and life-cycle.

3.3.1 Operational Assurance

Operational assurance includes the following aspects: system architecture, system integrity, covert channel analysis, trusted facility management, and trusted recovery.

3.3.1.1 System Architecture

The TCB must maintain a domain for its own execution that protects it from external interference or tampering. It must maintain process isolation through the provision of distinct address spaces under its control. The TCB must be internally structured into well-defined largely independent modules. It must make effective use of available hardware to separate those elements that are protection-critical from those that are not. TCB modules must be designed such that the principle of least privilege is enforced. Features in hardware, such as segmentation, must be used to support logically distinct storage objects with separate attributes (namely, readable and writable). The TCB user interface must be completely defined and all elements of the TCB must be identified. The TCB must be designed and structured to use a complete, conceptually simple protection mechanism with precisely defined semantics; this mechanism must play a central role in enforcing the internal structuring of the TCB and the system. The TCB must incorporate significant use of layering, data abstraction, and information hiding. Significant system engineering must be directed toward minimizing the complexity of the TCB and excluding from the TCB modules that are not protection-critical.

Guidelines for the use of Ada Constructs and Features:

The TCB's system architecture must be modular, and use data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types and packages. To ensure system integrity and to prevent the creation of covert channels, the creation of TCB system features (dynamic storage, input and output communications within the TCB and between the user/subject and the TCB, tasking and/or global and shared variables) must address the potential security problems associated with their implementation and use. The recommended constructs and their discussion in Section 2.0 are as follows:

ACCESS TYPES	2 - 3.8
INPUT and OUTPUT	2 - 14.
PACKAGES	2 - 7.
TYPES	2 - 3.

3.3.1.2 System Integrity

Hardware and/or software features must be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB.

Guidelines for the use of Ada Constructs and Features:

Take advantage of Ada to create readable code that helps the validation process. See the discussion of identifiers in Section 2 - 2.3.

3.3.1.3 Covert Channel Analysis

The search for covert channels required in this analysis is facilitated by having the TCB's software and its documentation be readable and understandable.

Guidelines for the use of Ada Constructs and Features:

Take advantage of Ada to create readable code which helps the identification of covert channels. Refer to the discussion of identifiers in Section 2 - 2.3.

3.3.1.4 Trusted Facility Management

The majority of issues related to trusted facility management are not software issues. Rather, they are concerned with the responsibilities of the security administrator and the ADP system administrative personnel.

Guidelines for the use of Ada Constructs and Features:

Take advantage of Ada to create readable code which helps the management of a trusted facility. Refer to the discussion of identifiers in Section 2-2.3.

3.3.1.5 Trusted Recovery

Procedures and/or mechanisms must be provided to assure that after an ADP system failure or other discontinuity, recovery without a protection compromise is obtained.

Guidelines for the use of Ada Constructs and Features:

Implement trusted recovery with Ada's exception handling mechanism, taking into account proper security considerations. Refer to the discussion of exception handling in Section 2 - 11. System reinitialization and backup should be handled with trusted recovery techniques. This requires the input and output operations involved in system reinitialization and backup to be trusted. Refer to the discussion of input and output in Section 2 - 14.

3.3.2 Life-Cycle Assurance

Life-Cycle assurance includes the following aspects: security testing, design specification and verification, and configuration management.

3.3.2.1 Security Testing

Security mechanisms of the ADP system must be tested and found to work as claimed in the system documentation. A team of individuals who thoroughly understand the specific implementation of the TCB must subject its design documentation, source code, and object code to thorough analysis and testing. The team's objective should be to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy. This will assure that no subject is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users. The TCB must be found resistant to penetration. All discovered flaws must be corrected, and the TCB must be retested to demonstrate that the flaws have been eliminated and that new flaws have not been introduced. Testing must demonstrate that the TCB implementation is consistent with the descriptive top-level specification. No design flaws and no more than a few correctable implementation flaws may be found during testing and there must be reasonable confidence that few remain.

Guidelines for the use of Ada Constructs and Features:

Security testing of the TCB system is promoted by having the system architecture exhibit modularity, and using data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types. Security testing of the TCB system must include the testing of all implementations of the following system features: dynamic storage management, input and output communications within the TCB and between the user/subject and the TCB, tasking and/or global and shared variables. The recommended constructs and their discussion in Section 2.0 are as follows:

INPUT and OUTPUT	2 - 14.
PACKAGES	2 - 7
SUBPROGRAMS	2 - 6.
TYPES	2 - 3.

3.3.2.2 Design Specification and Verification

A formal model of the security policy supported by the TCB must be maintained that is proven to be consistent with its axioms. A descriptive top-level specification (DTLS) of the TCB must be maintained that completely and accurately describes the TCB in terms of exceptions, error messages, and effects. It must be shown to be an accurate description of the TCB interface. A convincing argument must be given that the DTLS is consistent with the model.

3.3.2.3 Configuration Management

A configuration management system must be in place that maintains control of changes to the descriptive top-level specification, other design data, implementation documentation, source code, the running version of the object code, and test fixtures and documentation. The configuration management system must assure a consistent mapping among all documentation and code associated with the current version of the TCB. Tools must be provided for generation of a new version of the TCB from source code and for comparing a newly generated version with the previous TCB version in order to ascertain that only the intended changes have been made in the code that will actually be used as the new version of the TCB.

3.4 Documentation

Documentation is an important part of the software development process. It aids users who are not familiar with the system in learning how to use the system correctly. It aids support programmers in testing the system to ensure that a modification has not had a negative impact on the system. This is especially important when developing a TCB, because the security of the system is of the utmost importance. The documentation must be developed to meet all requirements set forth in the TCSEC, and must be complete and up to date.

Guidelines for the use of Ada Constructs and Features:

The documentation, particularly the design documentation, must clearly convey the implementation of the TCB system architecture, which must exhibit modularity, and use data abstraction with information hiding in its implementation. Ada is well suited to incorporate modularity with its packages and subprograms and to implement data abstraction and information hiding with abstract data types. These features not only aid the understandability of the code, but also of the design documentation. The documentation must clearly show how the security risks are handled, including those posed by the following TCB system features: dynamic storage management, input and output communications within the TCB and between the user/subject and the TCB, tasking and/or global and shared variables, and any tailoring or configuring of the Ada compiler or runtime system. This discussion of the application of Ada constructs to documentation applies to the following four subsections: Security Features User's Guide, Trusted Facility Manual, Test Documentation, and Design Documentation.

Ada coding conventions should use Ada's self-documenting capabilities, so that the code can contribute to the design and testing documentation. The self-documenting capabilities can be realized better with the use of the following conventions: Readable and understandable mnemonics should be used. The code should exhibit consistent indentation. Blank lines should be used to logically partition the code. Comments should be inserted into the code to provide information that is not conveyed by the code. Each package and subprogram should have a header that states its purpose, its authors, and the history (dates) of its creation and revision(s).

3.4.1 Security Features User's Guide

A single summary, chapter, or manual in user documentation must describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another.

Guidelines for the use of Ada Constructs and Features:

Ada will have no impact on the development of the Security Features User's Guide.

3.4.2 Trusted Facility Manual

A manual addressed to the ADP system administrator must present cautions about functions and privileges that should be controlled when running a secure facility. It must describe the operator and administrator functions related to security, to include changing the security characteristics of a user. The manual must provide guidelines on the consistent and effective use of the protection features of the system, how they interact, how to securely generate a new TCB, and facility procedures, warnings, and privileges that need to be controlled in order to operate the facility in a secure manner. TCB modules that contain the reference validation mechanism must be identified. Procedures for secure generation of a new TCB from source after modification of any modules in the TCB must be described. The manual must include the procedures to ensure that the system is initially started in a secure manner. Procedures must also be included to resume secure system operation after any lapse in system operation.

Guidelines for the use of Ada Constructs and Features:

Ada will have no impact on the development of the Trusted Facility Manual.

3.4.3 Test Documentation

The system developer must provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing. Documentation must include results of testing the effectiveness of the methods used to reduce covert channel bandwidths.

Guidelines for the use of Ada Constructs and Features:

Ada coding should be self-documenting, as discussed in Section 3 - 3.4, so that it can contribute to the testing documentation.

3.4.4 Design Documentation

Documentation must be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB. **The interfaces between the TCB modules must be described.** A formal description of the security policy model enforced by the TCB must be available and proven that is sufficient to enforce the security policy. Specific TCB protection mechanisms must be identified, and an explanation must be given to show that they satisfy the model. The TCB implementation (i.e., in hardware, firmware, and software) must be shown, using informal techniques, to be consistent with the DTLS. The elements of the DTLS must be shown, using informal techniques, to correspond to the elements of the TCB. Documentation must describe how the TCB implements the reference monitor concept and give an explanation why it is tamper resistant, cannot be bypassed, and is correctly implemented. Documentation must describe how the TCB is structured to facilitate testing and to enforce least privilege. Documentation must also present the results of covert channel analysis and the tradeoffs involved in restricting the channels. All auditable events that may be used in the exploitation of known covert storage channels must be identified. The bandwidths of known covert storage channels, the use of which is not detectable by the auditing mechanisms, must be provided.

Guidelines for the use of Ada Constructs and Features:

Ada code should be self-documenting, as discussed in Section 3.4, so that it can contribute to the design documentation. The use of Ada PDL will assist in generating readable and understandable design documentation.

4.0 SUMMARY AND CONCLUSIONS

4.1 SUMMARY

This appendix is a set of guidelines on how to use Ada in the development of TCBs. These include guidance on how to exploit the advantages of using Ada, such as data abstraction, information hiding, modularity, localization, strong data typing, packages, subprograms, and tasks. They also provide guidance on how to avoid problems with using Ada, which are associated with certain Ada constructs and features.

Sections 2.0 and 3.0 address the guidelines in two ways. First, Section 2.0 provides definitions of and general programming guidelines on the use of Ada constructs and features mapped directly to the LRM. Section 3.0 provides guidelines on the application of Ada constructs and features in the context of a mapping of Ada usage to generalized TCB criteria. In particular, Class B3 is used as a template for the generalized TCB criteria.

Those guidelines specific to both TCB development and Ada are presented here. A developer should be experienced with the TCB development process and have established guidelines for the development of TCBs. Also, a developer should use general-purpose Ada guidelines. The guidelines presented here are meant to complement both TCB development guidelines and general-purpose Ada guidelines.

4.2 CONCLUSIONS

Because Ada was designed with features and constructs that promote recognized sound software engineering principles, it is well suited as the implementation language of TCBs. Ada offers various specific benefits for the development of TCBs, such as the capability of creating TCB systems that exhibit modularity and information hiding with the use of packages. These guidelines are meant to enable developers to realize these benefits. Also, certain Ada constructs provide both advantages and disadvantages. In these cases the guidelines are meant to enable developers to realize the advantages of a construct while minimizing or eliminating its disadvantages.

5.0 COLLECTION OF EXAMPLES

The following examples have been collected from Sections 2.0 and 3.0 to provide a single area for referencing them. The section naming conventions "2 -" and "3 -" have been retained for consistency and ease of reference to the preceding sections.

2 - 2. Lexical Elements

Examples: This example illustrates underscoring in numeric literals, which improves the readability of literals that have many digits. Underscores are used in place of commas in large numbers. In strings of digits to the right of the decimal, an underscore is used, for example, after every fifth digit.

```
123_456 rather than 123456         -- integer literal
3.14159_26 rather than 3.1415926   -- real literal
```

2 - 2.4.2 Based Literals

Examples: This example illustrates base literals.

```
-- integer literals of value 255
2#1111_1111#     16#FF#     016#0FF#
```

2 - 2.3 Identifiers

Examples: The following examples illustrate clearly readable and understandable mnemonic identifier names. They are identifiers which are used in later examples in this appendix. The use of each identifier should be clear based on its name. This set of identifiers is taken out of context; thus they do not constitute compilable code.

```
Named_Object_ACL_Record         --  see 2 - 3.2.1
Max_Named_String_Length         --  see 2 - 3.2.2
Named_Individuals_List_Type     --  see 2 - 3.2.1
Scrubbed_Named_Objects_List     --  see 2 - 3.2.1
```

2 - 3.2.1 Object Declarations

Examples: The following examples illustrate clearly readable and understandable mnemonic object declarations. This set of object declarations include the initialization of the objects.

```
--      Max_Named_String_Length, Name_String_Type, and Blank_Name_String
--      are declared in package Basic_TCB_Types_Package. Because only essential
--      features of this package need to be shown, it is not included formally in
--      this appendix.

        Blank_Name_String : Name_String_Type                 -- see 2 - 3.6.3
          := ( 1 .. Max_Named_String_Length => ' ');         -- see 2 - 3.2.2


--      The next three declarations in this section are located in the array
--      types version of package Access_Control_List_Types_Package. Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        Scrubbed_Named_Objects_List :
                 Named_Objects_List_Type                     -- see 2 - 3.6
              := Named_Objects_List_Type'
                   ( others => Basic_TCB_Types_Package.
                                   Blank_Name_String );

        Scrubbed_Named_Individuals_List :
                 Named_Individuals_List_Type                 -- see 2 - 3.6
              := Named_Individuals_List_Type'
                   ( others => Basic_TCB_Types_Package.
                                   Blank_Name_String );

        Scrubbed_Groups_of_Named_Individuals_List :
                 Groups_of_Named_Individuals_List_Type       -- see 2 - 3.6
              := Groups_of_Named_Individuals_List_Type'
                   ( others => Basic_TCB_Types_Package.
                                   Blank_Name_String );


--      The next three declarations in this section are located in the access
--      types version of package Access_Control_List_Types_Package. Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.
```

```
        Scrubbed_Named_Objects_List :
                   Named_Objects_List_Type                     -- see 2 - 3.8
               := Named_Objects_List_Type'
                  ( Name => Basic_TCB_Types_Package.
                                Blank_Name_String,
                    Next => null );

        Scrubbed_Named_Individuals_List :
                   Named_Individuals_List_Type                 -- see 2 - 3.8
               := Named_Individuals_List_Type'
                  ( Name => Basic_TCB_Types_Package.
                                Blank_Name_String,
                    Next => null );


        Scrubbed_Groups_of_Named_Individuals_List :
                   Groups_of_Named_Individuals_List_Type       -- see 2 - 3.8
               := Groups_of_Named_Individuals_List_Type'
                  ( Name => Basic_TCB_Types_Package.
                                Blank_Name_String,
                    Next => null );

--      The remaining declarations in this section are located
--      in both the access and array types versions of
--      package Access_Control_List_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included formally in
--      this appendix.

        Scrubbed_Named_Object_ACL_Record :
          Named_Object_Record_Type                             -- see 2 - 3.7
            := Named_Object_Record_Type'
                ( Name => Basic_TCB_Types_Package.
                              Blank_Name_String,

                  Sensitivity_Label =>
                    Mandatory_Access_Control_Types_Package.
                      Scrubbed_Sensitivity_Label,               -- see 3 - 3.1.3

                  Authorized_Named_Individuals_List =>
                    Scrubbed_Named_Individuals_List,

                  Authorized_Groups_of_Named_Individuals_List =>
                    Scrubbed_Groups_of_Named_Individuals_List,

                  Unauthorized_Named_Individuals_List =>
                      Scrubbed_Named_Individuals_List );
```

```
Named_Object_ACL_Record : Named_Object_Record_Type   -- see 2 - 3.7
    := Scrubbed_Named_Object_ACL_Record;


Scrubbed_Named_Individual_ACL_Record :
  Named_Individual_Record_Type                       -- see 2 - 3.7
    := Named_Individual_Record_Type'
         ( Name              => Basic_TCB_Types_Package.
                                  Blank_Name_String,
                                                     -- see 2 - 3.6.3
           Sensitivity_Label =>
             Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label,            -- see 3 - 3.1.3

           Named_Objects_List => Scrubbed_Named_Objects_List );


Named_Individual_ACL_Record :
  Named_Individual_Record_Type                       -- see 2 - 3.7
    := Scrubbed_Named_Individual_ACL_Record;


Scrubbed_Group_of_Named_Individuals_ACL_Record :
  Group_of_Named_Individuals_Record_Type             -- see 2 - 3.7
    := Group_of_Named_Individuals_Record_Type'
         ( Name              => Basic_TCB_Types_Package.
                                  Blank_Name_String,
                                                     -- see 2 - 3.6.3
           Sensitivity_Label =>
             Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label,           -- see 3 - 3.1.3

           Named_Objects_List => Scrubbed_Named_Objects_List );


Group_of_Named_Individuals_ACL_Record :
  Group_of_Named_Individuals_Record_Type             -- see 2 - 3.7
    := Scrubbed_Group_of_Named_Individuals_ACL_Record;
```

2 - 3.2.2 Number Declarations

Example: The following example illustrates a clearly readable and understandable mnemonic number declaration

```
--      Max_Named_String_Length is declared in package Basic_TCB_Types_Package.
--      Because only essential features of this package need to be shown, it is
--      not included formally in this appendix.

        Max_Named_String_Length : constant := 80;
```

2 - 3.3.1 Type Declarations

Example: The following example illustrates a clearly readable and understandable mnemonic type declaration.

```
--      Day_Type is declared in package Basic_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        type Day_Type is ( Sunday, Monday, Tuesday, Wednesday,
                           Thursday, Friday, Saturday );
```

2 - 3.3.2 Subtype Declarations

Example: The following example illustrates a clearly readable and understandable mnemonic subtype declaration.

```
--      Weekday_Type is declared in package Basic_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        subtype Weekday_Type is Day_Type range Monday .. Friday;
```

2 - 3.4 Derived Types

Example: The following example illustrates a clearly readable and understandable mnemonic derived type declaration. The package Key_Manager_Package is declared in section 2 - 7.4.2.

```
type Access_Key_Type is new Key_Manager_Package. Key_Type;
-- the derived subprograms have the following specifications:

-- procedure Get_Key ( K : out Access_Key_Type );
-- function "<" ( X, Y : Access_Key_Type ) return BOOLEAN;
-- function "+" ( X, Y : Access_Key_Type ) return Access_Key_Type;
```

2 - 3.5.1 Enumeration Types

Example: The following example illustrates a clearly readable and understandable mnemonic enumeration type declaration.

```
--      Security_Classification_Type is declared in
--      package Basic_TCB_Types_Package.  Because only essential
--      features of this package need to be shown, it is not included
--      formally in this appendix.

        type Security_Classification_Type is
             ( Unclassified, Confidential, Secret, Top_Secret );
```

2 - 3.6 Array Types

Examples: The following examples illustrate clearly readable and understandable mnemonic array type declarations.

```
--      The following declarations in this section are located in the array
--      types version of package Access_Control_List_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        Max_Number_of_Objects                : constant NATURAL := 25;
        Max_Number_of_Individuals            : constant NATURAL := 25;
        Max_Number_of_Groups_of_Individuals  : constant NATURAL := 25;


        type Named_Objects_List_Type is array
             ( POSITIVE range 1 .. Max_Number_of_Objects ) of
               Basic_TCB_Types_Package. Name_of_Object_Type;    -- see 2 - 4.1

        type Named_Objects_ACL_Type is array
             ( POSITIVE range 1 .. Max_Number_of_Objects ) of
               Named_Object_Record_Type;                        -- see 2 - 3.2.1
```

```
type Named_Individuals_List_Type is array
     ( POSITIVE range 1 .. Max_Number_of_Individuals ) of
       Basic_TCB_Types_Package.
         Name_of_Individual_Type;                      --  see 2 - 4.1

type Named_Individuals_ACL_Type is array
     ( POSITIVE range 1 .. Max_Number_of_Individuals ) of
       Named_Individual_Record_Type;                   --  see 2 - 3.2.1


type Groups_of_Named_Individuals_List_Type is array
     ( POSITIVE range 1 .. Max_Number_of_Groups_of_Individuals ) of
       Basic_TCB_Types_Package.
         Name_of_Group_of_Individuals_Type;            -- see 2 - 4.1

type Groups_of_Named_Individuals_ACL_Type is array
     ( POSITIVE range 1 .. Max_Number_of_Groups_of_Individuals ) of
       Group_of_Named_Individuals_Record_Type;         --  see 2 - 3.2.1
```

2 - 3.6.3 The Type String

Example: The following example illustrates a clearly readable and understandable mnemonic string type declaration.

```
--      Name_String_Type is declared in package Basic_TCB_Types_Package.
--      Because only essential features of this package need to be shown,
--      it is not included formally in this appendix.

        subtype Name_String_Type is
               STRING ( 1 .. Max_Named_String_Length );    --  see 2 - 3.2.2
```

2 - 3.7 Record Types

Examples: The following examples illustrate clearly readable and understandable mnemonic record type declarations.

```
--      The following declarations in this section are located in both the access
--      and array types versions of package Access_Control_List_Types_Package.
--      Because only essential features of this package need to be shown, it is
--      not included formally in this appendix.
```

```
type Named_Object_Record_Type is
  record
    Name : Basic_TCB_Types_Package. Name_of_Object_Type  --  see 2 - 4.1
      := Basic_TCB_Types_Package. Blank_Name_String;   --  see 2 - 3.2.1

    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;             --  see 3 - 3.1.3

    Authorized_Named_Individuals_List :
               Named_Individuals_List_Type     --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Individuals_List;      --  see 2 - 3.2.1

    Authorized_Groups_of_Named_Individuals_List :
                                               --  see 2 - 3.6 and 2 - 3.8
               Groups_of_Named_Individuals_List_Type
      := Scrubbed_Groups_of_Named_Individuals_List;  --  see 2 - 3.2.1

    Unauthorized_Named_Individuals_List :
                Named_Individuals_List_Type    --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Individuals_List;      --  see 2 - 3.2.1
end record;


type Named_Individual_Record_Type is
  record
    Name : Basic_TCB_Types_Package.
             Name_of_Individual_Type                  --  see 2 - 4.1
      := Basic_TCB_Types_Package. Blank_Name_String;   --  see 2 - 3.2.1

    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;             --  see 3 - 3.1.3

    Named_Objects_List :
      Named_Objects_List_Type              --  see 2 - 3.6 and 2 - 3.8
      := Scrubbed_Named_Objects_List;      --  see 2 - 3.2.1
end record;
```

```
type Group_of_Named_Individuals_Record_Type is
  record
    Name : Basic_TCB_Types_Package.
             Name_of_Group_of_Individuals_Type        -- see 2 - 4.1
      := Basic_TCB_Types_Package. Blank_Name_String;  -- see 2 - 3.2.1


    Sensitivity_Label :
      Mandatory_Access_Control_Types_Package.
        Sensitivity_Label_Type
          := Mandatory_Access_Control_Types_Package.
               Scrubbed_Sensitivity_Label;           -- see 3 - 3.1.3

    Named_Objects_List :
      Named_Objects_List_Type             -- see 2 - 3.6 and 2 - 3.8
        := Scrubbed_Named_Objects_List;   -- see 2 - 3.2.1
end record;
```

2 - 3.8 Access Types

Examples: The following examples illustrate clearly readable and understandable mnemonic access type declarations. To ensure that the memory allocated to a node is scrubbed, the components of a node's record type should be initialized to scrubbed values, as shown below. Also, ensure that a node record is set to scrubbed values before it is placed back in the heap, i.e., before nothing points to it any longer.

```
--      The following declarations in this section are located in the access
--      types version of package Access_Control_List_Types_Package.  Because only
--      essential features of this package need to be shown, it is not included
--      formally in this appendix.

        type Named_Objects_List_Type;
        type Named_Objects_List_Pointer_Type
          is access Named_Objects_List_Type;

        type Named_Objects_List_Type is
          record
            Name : Basic_TCB_Types_Package.
                     Name_of_Object_Type                -- see 2 - 4.1
              := Basic_TCB_Types_Package.
                   Blank_Name_String;                   -- see 2 - 3.2.1
            Next : Named_Objects_List_Pointer_Type := null;
        end record;
```

```
type Named_Objects_ACL_Type;
type Named_Objects_ACL_Pointer_Type
  is access Named_Objects_ACL_Type;

type Named_Objects_ACL_Type is
  record
    Named_Object_ACL_Record :
      Named_Object_Record_Type
      := Scrubbed_Named_Object_ACL_Record;          -- see 2 - 3.2.1

    Next : Named_Objects_ACL_Pointer_Type
      := null;
end record;


type Named_Individuals_List_Type
type Named_Individuals_List_Pointer_Type
  is access Named_Individuals_List_Type;

type Named_Individuals_List_Type is
  record
    Name : Basic_TCB_Types_Package.
               Name_of_Individual_Type              -- see 2 - 4.1
      := Basic_TCB_Types_Package.
           Blank_Name_String;                       -- see 2 - 3.2.1

    Next : Named_Individuals_List_Pointer_Type := null;
end record;


type Named_Individuals_ACL_Type;
type Named_Individuals_ACL_Pointer_Type
  is access Named_Individuals_ACL_Type;

type Named_Individuals_ACL_Type is
  record
    Named_Individual_ACL_Record :
      Named_Individual_Record_Type
    := Scrubbed_Named_Individual_ACL_Record;        -- see 2 - 3.2.1

    Next : Named_Individuals_ACL_Pointer_Type
      := null;
end record;
```

```
    type Groups_of_Named_Individuals_List_Type
    type Groups_of_Named_Individuals_List_Pointer_Type
      is access Groups_of_Named_Individuals_List_Type;

    type Groups_of_Named_Individuals_List_Type is
      record
        Name : Basic_TCB_Types_Package.
                   Name_of_Group_of_Individuals_Type;      -- see 2 - 4.1
          := Basic_TCB_Types_Package.
                 Blank_Name_String;                        -- see 2 - 3.2.1

        Next : Groups_of_Named_Individuals_List_Pointer_Type
          := null;
    end record;


    type Groups_of_Named_Individuals_ACL_Type;
    type Groups_of_Named_Individuals_ACL_Pointer_Type
      is access Groups_of_Named_Individuals_ACL_Type;

    type Groups_of_Named_Individuals_ACL_Type is
      record
        Group_of_Named_Individuals_ACL_Record :
          Group_of_Named_Individuals_Record_Type
          := Scrubbed_Group_of_Named_Individuals_ACL_Record;
                                                                 -- see 2 - 3.2.1
        Next : Groups_of_Named_Individuals_ACL_Pointer_Type
          := null;
    end record;
```

2 - 4.1 Names

Examples: The following examples illustrate clearly readable and understandable mnemonic names.

```
--      The following type declarations in this section are located in
--      package Basic_TCB_Types_Package.  Because only essential features of this
--      package need to be shown, it is not included formally in this appendix.

        subtype Name_of_Object_Type is Name_String_Type;      -- see 2 - 3.2.2

        Name_of_Object : Name_of_Object_Type
          := Blank_Name_String;                                -- see 2 - 3.2.1
```

```
subtype Name_of_Individual_Type is Name_String_Type;  -- see 2 - 3.2.2

Name_of_Individual : Name_of_Individual_Type
  := Blank_Name_String;                                -- see 2 - 3.2.1


subtype Name_of_Group_of_Individuals_Type is Name_String_Type;
                                                       -- see 2 - 3.2.2
Name_of_Group_of_Named_Individuals :
  Name_of_Group_of_Individuals_Type
    := Blank_Name_String;                              -- see 2 - 3.2.1
```

2 - 6.4.2 Default Parameters

Example: This example illustrates named parameter association.

```
procedure Search_File ( File : in out
                        Access_Control_List_Types_Package.
                          ACL_File_Type;
                         Key   : in     Name;
                         Index :    out File_Index );

Search_File ( File  => ACL_File,
              Key   => "Smith , J",
              Index => Record_Entry );
```

Example: This example from package TEXT_IO illustrates default parameters.

```
procedure CREATE ( FILE : in out FILE_TYPE;
                   MODE : in     FILE_MODE := OUT_FILE;
                   NAME : in     STRING    := "";
                   FORM : in     STRING    := "" );

TEXT_IO.  CREATE  (  FILE  =>
Named_Objects_Access_Control_List_Manager_Package.
          ACL_File );
```

2 - 7.2 Package Specifications and Declarations

Example: This example illustrates data abstraction and information hiding, modularity, and localization achieved with package specification and declarations.

```
with Basic_TCB_Types_Package;
with Access_Control_List_Types_Package;
package Named_Objects_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_Named_Object_ACL_Record
      ( Name_of_Object : in     Basic_TCB_Types_Package.
                                  Name_of_Object_Type;     -- see 2 - 4.1
        Named_Object_ACL_Record :   out
          Access_Control_List_Types_Package.
            Named_Object_Record_Type );                    -- see 2 - 4.1

    procedure Insert_Named_Object_ACL_Record
      ( Named_Object_ACL_Record : in
          Access_Control_List_Types_Package.
            Named_Object_Record_Type );                    -- see 2 - 4.1

    procedure Delete_Named_Object_ACL_Record
      ( Name_of_Object : in Basic_TCB_Types_Package.
                              Name_of_Object_Type );       -- see 2 - 4.1

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;

  private

    -- . . .

end Named_Objects_Access_Control_List_Manager_Package;


package body Named_Objects_Access_Control_List_Manager_Package is

    -- By declaring the Named_Objects_ACL (see 2 - 12.4) in the
    -- body of this package rather than in the specification,
    -- it is hidden from the user of this package.  Thus, the user
    -- can gain only indirect access to it through the subprograms
    -- declared in the specification.  The typical list manipulation
    -- operations ( e.g., as illustrated by Booch 1987A and
    -- Feldman 1985 ) are only provided in the package body.

    -- . . .
-- Typical list manipulation operations

-- . . .

procedure Get_Named_Object_ACL_Record
  ( Name_of_Object : in     Basic_TCB_Types_Package.
                              Name_of_Object_Type;    -- see 2 - 4.1
    Named_Object_ACL_Record :   out
      Access_Control_List_Types_Package.
        Named_Object_Record_Type ) is                 -- see 2 - 4.1

    -- . . .

  begin  -- Get_Named_Object_ACL_Record

    -- . . .

    -- Sequence through the Named_Object_ACL using the typical
    -- list manipulation operations to locate and get the
    -- indicated Named_Object_ACL_Record.

    -- . . .

end Get_Named_Object_ACL_Record;

procedure Insert_Named_Object_ACL_Record
  ( Named_Object_ACL_Record : in
      Access_Control_List_Types_Package.
        Named_Object_Record_Type ) is                 -- see 2 - 4.1

    -- . . .

  begin  -- Insert_Named_Object_ACL_Record

    -- . . .

    -- Sequence through the Named_Object_ACL using the typical
    -- list manipulation operations to locate the appropriate
    -- place to insert the indicated Named_Object_ACL_Record.
    -- This location is determined by a predefined mechanism,
    -- e.g., alphabitizing by the Named_Object_ACL_Record.Name,
    -- or more crudely by a simple (FIFO) stack or (FILO) queue.

    -- . . .

end Insert_Named_Object_ACL_Record;
```

```
        procedure Delete_Named_Object_ACL_Record
          ( Name_of_Object : in Basic_TCB_Types_Package.
                                  Name_of_Object_Type ) is   --  see 2 - 4.1

            -- . . .

          begin  -- Delete_Named_Object_ACL_Record

            -- . . .

            -- Sequence through the Named_Object_ACL using the typical
            -- list manipulation operations to locate, scrub, and delete
            -- the indicated Named_Object_ACL_Record.

            -- . . .

        end Delete_Named_Object_ACL_Record;

        -- . . .

    end Named_Objects_Access_Control_List_Manager_Package;
```

2 - 7.4.2 Operations of a Private Type

Example: The abstract data type, Key_Type, is created with the use of the package Key_Manager_Package and its private type, Key_Type.

```
    package Key_Manager_Package is
        type Key_Type is private;
        Null_Key : constant Key_Type;
        procedure Get_Key ( K : out Key_Type );
        function "<" ( X, Y : Key_Type ) return BOOLEAN;
        function "+" ( X, Y : Key_Type ) return Key_Type;

      private

        type Key_Type is new NATURAL;
        Null_Key : constant Key_Type := 0;
    end Key_Manager_Package;
```

```
package body Key_Manager_Package is

    Last_Key : Key_Type := 0;

    procedure Get_Key ( K : out Key_Type ) is
      begin
        Last_Key := Last_Key + 1;
        K := Last_Key;
    end Get_Key;

    function "<" ( X, Y : Key_Type ) return BOOLEAN is
      begin
        return NATURAL ( X ) < NATURAL ( Y );
    end "<";

    function "+" ( X, Y : Key_Type ) return Key_Type is
      begin
        return Key_Type ( NATURAL ( X ) + NATURAL ( Y ) );
    end "+";

end Key_Manager_Package;
```

2 - 9.2 Task Types and Task Objects

Example: This example illustrates trusted generalized mechanisms to control and regulate intertask communications with semaphores.

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;        -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                     -- see 3 - 3.2.3
generic

  Max_Number_of_Tasks_Allowed : in NATURAL := 1;

package Generic_Counting_Semaphore_Manager_Package is

  task type Counting_Semaphore_Task_Type is
    entry Allow_Task_to_Pass (
            Other_Task_MAC_Record : in
              Mandatory_Access_Control_Types_Package.
                MAC_Record_Type );                    -- see 3 - 3.1.3
```

```
      entry Release_Task (
              Other_Task_MAC_Record : in
                Mandatory_Access_Control_Types_Package.
                  MAC_Record_Type );                  -- see 3 - 3.1.3

    end Counting_Semaphore_Task_Type;

    -- . . .

  end Generic_Counting_Semaphore_Manager_Package;


  package body Generic_Counting_Semaphore_Manager_Package is

    task body Counting_Semaphore_Task_Type is

        Number_of_Tasks : INTEGER := Max_Number_of_Tasks_Allowed;

        Local_MAC_Record :
          Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
            MAC_Record_Type;

        Other_Task_MAC_Record :
          Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
            MAC_Record_Type;

      begin   -- Counting_Semaphore_Task_Type
        loop
          select
              when Number_of_Tasks > 0 =>
                accept Allow_Task_to_Pass (
                         Other_Task_MAC_Record : in
                           Mandatory_Access_Control_Types_Package.
                             MAC_Record_Type ) do    -- see 3 - 3.1.3

                  if Mandatory_Access_Control_Manager_Package.
                       Sensitivity_Labels_Match       -- see 3 - 3.1.3
                         ( Local_MAC_Record. Sensitivity_Label,
                           Other_Task_MAC_Record.
                             Sensitivity_Label ) then

                    Audit_Trail_Manager_Package.      -- see 3 - 3.2.3
                      Log_Subjects_Access_to_Object_in_Audit_Trail (
                        Local_MAC_Record, Other_Task_MAC_Record );
                  Number_of_Tasks := Number_of_Tasks - 1;
                end if;
              end Allow_Task_to_Pass;

          or
            when Number_of_Tasks < Max_Number_of_Tasks_Allowed
              => accept Release_Task (
                         Other_Task_MAC_Record : in
                           Mandatory_Access_Control_Types_Package.
                             MAC_Record_Type ) do  --  see 3 - 3.1.3

                   if Mandatory_Access_Control_Manager_Package.
                        Sensitivity_Labels_Match       --  see 3 - 3.1.3
                          ( Local_MAC_Record. Sensitivity_Label,
                            Other_Task_MAC_Record.
                              Sensitivity_Label ) then

                     Audit_Trail_Manager_Package.    --  see 3 - 3.2.3
                       Log_Subjects_Access_to_Object_in_Audit_Trail (
                         Local_MAC_Record, Other_Task_MAC_Record );

                     Number_of_Tasks := Number_of_Tasks + 1;
                   end if;
                 end Release_Task;
         end select;
       end loop;
     end Counting_Semaphore_Task_Type;

     -- . . .

   end Generic_Counting_Semaphore_Manager_Package;
```

2 - 9.12 Example of Tasking

Example: This example illustrates trusted generalized mechanisms to control and regulate intertask communications with mailboxes.

```
   with Mandatory_Access_Control_Types_Package;         --  see 3 - 3.1.3
   generic

     type Message_Type is private;

     Max_Number_of_Messages : in NATURAL := 24;
```

```
package Generic_Mailbox_Manager_Package is

  procedure Send    ( Message          : in  Message_Type;
                      Local_MAC_Record : in
                        Mandatory_Access_Control_Types_Package.
                          MAC_Record_Type );         --  see 3 - 3.1.3

  procedure Receive ( Message          :    out Message_Type;
                      Local_MAC_Record : in
                        Mandatory_Access_Control_Types_Package.
                          MAC_Record_Type );         --  see 3 - 3.1.3

end Generic_Mailbox_Manager_Package;


with Mandatory_Access_Control_Manager_Package;       --  see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    --  see 3 - 3.2.3
package body Generic_Mailbox_Manager_Package is

  task Manager_Task is

    entry Deposit ( Message                 : in Message_Type;
                    Other_Task_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type );           --  see 3 - 3.1.3

    entry Remove  ( Message                 :    out Message_Type;
                    Other_Task_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type );           --  see 3 - 3.1.3
  end Manager_Task;


  procedure Send ( Message          : in  Message_Type;
                   Local_MAC_Record : in
                     Mandatory_Access_Control_Types_Package.
                       MAC_Record_Type ) is          --  see 3 - 3.1.3
    begin
      Manager_Task. Deposit ( Message, Local_MAC_Record );
  end Send;


  procedure Receive ( Message          :   out Message_Type;
                      Local_MAC_Record : in
                        Mandatory_Access_Control_Types_Package.
                          MAC_Record_Type ) is       --  see 3 - 3.1.3
    begin
      Manager_Task. Remove ( Message, Local_MAC_Record );
  end Receive;


  task body Manager_Task is

      subtype Mailbox_Slot_Index_Type is INTEGER range
        0 .. ( Max_Number_of_Messages - 1 );

      Head_Slot : Mailbox_Slot_Index_Type := 0;
      Tail_Slot : Mailbox_Slot_Index_Type := 0;

      Message_Number : INTEGER range 0 .. Max_Number_of_Messages;

      Mailbox : array ( Mailbox_Slot_Index_Type ) of Message_Type;

      Local_MAC_Record :
        Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
          MAC_Record_Type;

      Other_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
          MAC_Record_Type;

    begin
      loop
        select
            when Message_Number < Max_Number_of_Messages =>
              accept Deposit ( Message                : in Message_Type;
                               Other_Task_MAC_Record : in
                                 Mandatory_Access_Control_Types_Package.
                                   MAC_Record_Type ) do
                                                       -- see 3 - 3.1.3

                if Mandatory_Access_Control_Manager_Package.
                      Sensitivity_Labels_Match         -- see 3 - 3.1.3
                        ( Local_MAC_Record. Sensitivity_Label,
                          Other_Task_MAC_Record.
                            Sensitivity_Label ) then

                  Audit_Trail_Manager_Package.         -- see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_MAC_Record, Other_Task_MAC_Record );
                  Mailbox ( Head_Slot ) := Message;

                  Head_Slot := ( Head_Slot + 1 ) mod
                                 Max_Number_of_Messages;

                  Message_Number := Message_Number + 1;
                end if;
              end Deposit;

          or
            when Message_Number > 0 =>
              accept Remove ( Message              : out Message_Type;
                              Other_Task_MAC_Record : in
                                Mandatory_Access_Control_Types_Package.
                                  MAC_Record_Type ) do
                                                    -- see 3 - 3.1.3

                if Mandatory_Access_Control_Manager_Package.
                     Sensitivity_Labels_Match          -- see 3 - 3.1.3
                       ( Local_MAC_Record. Sensitivity_Label,
                         Other_Task_MAC_Record.
                           Sensitivity_Label ) then


                  Audit_Trail_Manager_Package.       -- see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_MAC_Record, Other_Task_MAC_Record );

                  Message := Mailbox ( Head_Slot );

                  Tail_Slot := ( Tail_Slot + 1 ) mod
                                 Max_Number_of_Messages;

                  Message_Number := Message_Number - 1;
                end if;
              end Remove;
        end select;
      end loop;
  end Manager_Task;

end Generic_Mailbox_Manager_Package;
```

2 - 11. Exceptions

Example: This example illustrates trusted exception handling that allows program execution to continue in a trusted manner.

```
generic

  -- . . .

  type Name_Type is private;
  type ACL_Record_Type is private;

  -- . . .

package Generic_Access_Control_List_Manager_Package is

    -- . . .

    procedure Get_ACL_Record
      ( Name       : in      Name_Type;
        ACL_Record :     out ACL_Record_Type );

    procedure Insert_ACL_Record
      ( ACL_Record : in ACL_Record_Type  );

    procedure Delete_ACL_Record
      ( Name : in Name_Type );

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;

  private

    -- . . .

end Generic_Access_Control_List_Manager_Package;


with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;         -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;       -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    -- see 3 - 3.2.3
package body Generic_Access_Control_List_Manager_Package is
```

```
-- The user can gain only indirect access to instantiated
-- access control list through the subprograms declared in the
-- package specification.  Thus the access control list data
-- structure is hidden from the user of this package.  The
-- typical list manipulation operations ( e.g., as illustrated
-- by Booch 1987A and Feldman 1985 ) are only provided in the
-- package body.

-- . . .

Exception_Raiser_Record:
  Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
    MAC_Record_Type;          -- Initialize Exception_Raiser_Record.

Exception_Handler_Record :
  Mandatory_Access_Control_Types_Package.        -- see 3 - 3.1.3
    MAC_Record_Type;          -- Initialize Exception_Handler_Record.

-- . . .

-- Typical list manipulation operations

-- . . .

procedure Get_ACL_Record
  ( Name       : in     Name_Type;
    ACL_Record :    out ACL_Record_Type ) is

    -- . . .

    Exception_Name :
      Mandatory_Access_Control_Types_Package.    -- see 3 - 3.1.3
        Exception_Name_Type :=
          Mandatory_Access_Control_Types_Package.
            Others_String;

    -- . . .


  begin  -- Get_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- and get the indicated access control list record.
    -- . . .

  exception

    -- . . .

    when others =>
           Audit_Trail_Manager_Package.          -- see 3 - 3.2.3
             Log_Exception_in_Audit_Trail (
               Exception_Name,
               Exception_Raiser_Record,
               Exception_Handler_Record );

end Get_ACL_Record;


procedure Insert_ACL_Record
  ( ACL_Record : in ACL_Record_Type ) is

    -- . . .

    Exception_Name :
      Mandatory_Access_Control_Types_Package.    -- see 3 - 3.1.3
        Exception_Name_Type :=
          Mandatory_Access_Control_Types_Package.
            Others_String;

    -- . . .

  begin  -- Insert_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- the appropriate place to insert the indicated
    -- access control list record.  This location is
    -- determined by a predefined mechanism, e.g., alphabitizing by
    -- the "name" of the access control list record, or more
    -- crudely by a simple (FIFO) stack or (FILO) queue.

    -- . . .

    -- Check for the exception Overflow_Access_Control_List.
    -- If the exception is to be raised, then set the
    -- Exception_Name and Exception_Raiser_Record.

    -- . . .

  exception

    -- . . .

    when Overflow_Access_Control_List =>
           Audit_Trail_Manager_Package.          --  see 3 - 3.2.3
             Log_Exception_in_Audit_Trail (
               Exception_Name,
               Exception_Raiser_Record,
               Exception_Handler_Record );

    -- . . .

    when others =>
           Audit_Trail_Manager_Package.          --  see 3 - 3.2.3
             Log_Exception_in_Audit_Trail (
               Exception_Name,
               Exception_Raiser_Record,
               Exception_Handler_Record );

end Insert_ACL_Record;

procedure Delete_ACL_Record ( Name : in Name_Type ) is

    -- . . .

    Exception_Name :
      Mandatory_Access_Control_Types_Package.    --  see 3 - 3.1.3
        Exception_Name_Type :=
          Mandatory_Access_Control_Types_Package.
            Others_String;

    -- . . .

  begin  -- Delete_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to
    -- locate, scrub, and delete the indicated
```

```
          -- access control list record.

          -- . . .

          -- Check for the exception Access_Control_List_is_Null.
          -- If the exception is to be raised, then set the
          -- Exception_Name and Exception_Raiser_Record.

          -- . . .

        exception

          -- . . .

          when Access_Control_List_is_Null =>
                 Audit_Trail_Manager_Package.          -- see 3 - 3.2.3
                   Log_Exception_in_Audit_Trail (
                     Exception_Name,
                     Exception_Raiser_Record,
                     Exception_Handler_Record );

          -- . . .

          when others =>
                 Audit_Trail_Manager_Package.          -- see 3 - 3.2.3
                   Log_Exception_in_Audit_Trail (
                     Exception_Name,
                     Exception_Raiser_Record,
                     Exception_Handler_Record );

      end Delete_ACL_Record;

      -- . . .

    end Generic_Access_Control_List_Manager_Package;
```

2 - 12.4 Example of a Generic Package

Example: This example illustrates a generic package for multiple instances of an access control list, with instantiations of the package.

```
    generic

      -- . . .
```

```
   type Name_Type is private;
   type ACL_Record_Type is private;

   -- . . .

package Generic_Access_Control_List_Manager_Package is

    -- . . .


    procedure Get_ACL_Record
      ( Name       : in     Name_Type;
        ACL_Record :    out ACL_Record_Type );

    procedure Insert_ACL_Record
      ( ACL_Record : in ACL_Record_Type );

    procedure Delete_ACL_Record
      ( Name : in Name_Type );

    -- . . .

    Overflow_Access_Control_List : exception;
    Access_Control_List_is_Null  : exception;

  private

    -- . . .

end Generic_Access_Control_List_Manager_Package;


with Access_Control_List_Types_Package;
with Mandatory_Access_Control_Types_Package;         -- see 3 - 3.1.3
with Mandatory_Access_Control_Manager_Package;       -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    -- see 3 - 3.2.3
package body Generic_Access_Control_List_Manager_Package is

    -- The user can gain only indirect access to instantiated
    -- access control list through the subprograms declared in the
    -- package specification.  Thus the access control list data
    -- structure is hidden from the user of this package.  The
    -- typical list manipulation operations ( e.g., as illustrated
    -- by Booch 1987A and Feldman 1985 ) are only provided in the
    -- package body.
-- . . .

-- Typical list manipulation operations

-- . . .

procedure Get_ACL_Record
  ( Name       : in     Name_Type;
    ACL_Record :    out ACL_Record_Type ) is

    -- . . .

  begin  -- Get_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- and get the indicated access control list record.

    -- . . .

end Get_ACL_Record;

procedure Insert_ACL_Record
  ( ACL_Record : in ACL_Record_Type ) is

    -- . . .

  begin  -- Insert_ACL_Record

    -- . . .

    -- Sequence through the access control list data structure
    -- using the typical list manipulation operations to locate
    -- the appropriate place to insert the indicated
    -- access control list record.  This location is
    -- determined by a predefined mechanism, e.g., alphabitizing by
    -- the "name" of the access control list record, or more
    -- crudely by a simple (FIFO) stack or (FILO) queue.

    -- . . .

end Insert_ACL_Record;
```

```
    procedure Delete_ACL_Record ( Name : in Name_Type ) is

       -- . . .

     begin  -- Delete_ACL_Record

       -- . . .

       -- Sequence through the access control list data structure
       -- using the typical list manipulation operations to
       -- locate, scrub, and delete the indicated
       -- access control list record.

       -- . . .

    end Delete_ACL_Record;

    -- . . .

end Generic_Access_Control_List_Manager_Package;

-- Instantiations of Generic_Access_Control_List_Manager_Package
package Named_Objects_Access_Control_List_Manager_Package is new
        Generic_Access_Control_List_Manager_Package
          ( Name_Type       => Basic_TCB_Types_Package.
                                 Name_of_Object_Type,
            ACL_Record_Type => Access_Control_List_Types_Package.
                                 Named_Object_Record_Type );


package Named_Individuals_Access_Control_List_Manager_Package is new
        Generic_Access_Control_List_Manager_Package
          ( Name_Type       => Basic_TCB_Types_Package.
                                 Name_of_Object_Type,
            ACL_Record_Type => Access_Control_List_Types_Package.
                                 Named_Individual_Record_Type );
```

```
package Groups_of_Named_Individuals_ACL_Manager_Package is new
        Generic_Access_Control_List_Manager_Package
          ( Name_Type       => Basic_TCB_Types_Package.
                                 Name_of_Object_Type,
            ACL_Record_Type => Access_Control_List_Types_Package.
                               Group_of_Named_Individuals_Record_Type );
```

2 - 13.5.1 Interrupts

Example: This example illustrates trusted interrupt handling.
Note that address clauses are not currently supported in VAX Ada.

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
package Interrupt_Handler_Package is

  procedure Get_Character ( Char             :   out CHARACTER;
                            Local_MAC_Record : in
                              Mandatory_Access_Control_Types_Package.
                                MAC_Record_Type );   -- see 3 - 3.1.3

  procedure Put_Character ( Char             : in CHARACTER;
                            Local_MAC_Record : in
                              Mandatory_Access_Control_Types_Package.
                                MAC_Record_Type );   -- see 3 - 3.1.3

end Interrupt_Handler_Package;


with Mandatory_Access_Control_Manager_Package;        -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                     -- see 3 - 3.2.3
package body Interrupt_Handler_Package is

  -- . . .

  task Interrupt_Input_Handler_Task is

    pragma PRIORITY ( 4 );
         -- must have at least the priority of the interrupt

    entry Get_Character_from_Interrupt_Input_Address
            ( Char                   : out CHARACTER;
              Other_Task_MAC_Record : in
                Mandatory_Access_Control_Types_Package.
                  MAC_Record_Type );                  -- see 3 - 3.1.3
  entry Save_Hardware_Buffer_Character (
          Other_Task_MAC_Record : in
            Mandatory_Access_Control_Types_Package.
              MAC_Record_Type );                  -- see 3 - 3.1.3

  -- assuming that SYSTEM. ADDRESS is an INTEGER type
  for Save_Hardware_Buffer_Character use at 16#0020#;

end Interrupt_Input_Handler_Task;


task Interrupt_Output_Handler_Task is

  pragma PRIORITY ( 4 );
       -- must have at least the priority of the interrupt

  entry Deposit_Character_into_Hardware_Buffer (
          Other_Task_MAC_Record : in
            Mandatory_Access_Control_Types_Package.
              MAC_Record_Type );                  -- see 3 - 3.1.3

  entry Put_Character_into_Interrupt_Output_Address (
          Char                  : in CHARACTER;
          Other_Task_MAC_Record : in
            Mandatory_Access_Control_Types_Package.
              MAC_Record_Type );                  -- see 3 - 3.1.3

  -- assuming that SYSTEM. ADDRESS is an INTEGER type
  for Deposit_Character_into_Hardware_Buffer use at 16#0024#;

end Interrupt_Output_Handler_Task;


procedure Get_Character ( Char             :    out CHARACTER;
                          Local_MAC_Record : in
                            Mandatory_Access_Control_Types_Package.
                              MAC_Record_Type ) is

                                                  -- see 3 - 3.1.3
  begin  -- Get_Character

    Interrupt_Input_Handler_Task.
      Get_Character_from_Interrupt_Input_Address
        ( Char, Local_MAC_Record  );

end Get_Character;
```

```
procedure Put_Character ( Char             : in CHARACTER;
                          Local_MAC_Record : in
                            Mandatory_Access_Control_Types_Package.
                              MAC_Record_Type ) is
                                                      -- see 3 - 3.1.3
  begin  -- Put_Character

    Interrupt_Output_Handler_Task.
      Put_Character_into_Interrupt_Output_Address
        ( Char, Local_MAC_Record );

end Put_Character;


task body Interrupt_Input_Handler_Task is

    Max_Size_of_Internal_Input_Buffer : constant POSITIVE
      := 64;

    Internal_Input_Buffer :
      array ( 1 .. Max_Size_of_Internal_Input_Buffer )
        of CHARACTER;

    Input_Buffer_Pointer  : POSITIVE := 1;
    Output_Buffer_Pointer : POSITIVE := 1;

    Buffer_Count : INTEGER := 1;

    Hardware_Character_Buffer : CHARACTER;
    for Hardware_Character_Buffer use at 16#0100#;

    Local_Input_Task_MAC_Record :
      Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
        MAC_Record_Type;

    Other_Task_MAC_Record :
      Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
        MAC_Record_Type;

  begin  -- Interrupt_Input_Handler_Task
    loop
      select
        when Buffer_Count > 0 =>
        accept Get_Character_from_Interrupt_Input_Address
                ( Char                     :     out CHARACTER;
                  Other_Task_MAC_Record : in
                    Mandatory_Access_Control_Types_Package.
                      MAC_Record_Type ) do
                                                  -- see 3 - 3.1.3

          if Mandatory_Access_Control_Manager_Package.
                Sensitivity_Labels_Match          -- see 3 - 3.1.3
                  ( Local_Input_Task_MAC_Record.
                      Sensitivity_Label,
                    Other_Task_MAC_Record.
                      Sensitivity_Label ) then

            Audit_Trail_Manager_Package.         -- see 3 - 3.2.3
              Log_Subjects_Access_to_Object_in_Audit_Trail (
                Local_Input_Task_MAC_Record,
                Other_Task_MAC_Record );

            Char := Internal_Input_Buffer(
                      Output_Buffer_Pointer );

            Output_Buffer_Pointer :=
              Output_Buffer_Pointer mod
                Max_Size_of_Internal_Input_Buffer + 1;

            Buffer_Count := Buffer_Count - 1;
          end if;
        end Get_Character_from_Interrupt_Input_Address;

    or
      when Buffer_Count <
             Max_Size_of_Internal_Input_Buffer =>

        accept Save_Hardware_Buffer_Character (
                  Other_Task_MAC_Record : in
                    Mandatory_Access_Control_Types_Package.
                      MAC_Record_Type ) do      -- see 3 - 3.1.3

          if Mandatory_Access_Control_Manager_Package.
                Sensitivity_Labels_Match          -- see 3 - 3.1.3
                  ( Local_Input_Task_MAC_Record.
                      Sensitivity_Label,
                    Other_Task_MAC_Record.
                      Sensitivity_Label ) then
                  Audit_Trail_Manager_Package.      -- see 3 - 3.2.3
                    Log_Subjects_Access_to_Object_in_Audit_Trail (
                      Local_Input_Task_MAC_Record,
                      Other_Task_MAC_Record );

                  Internal_Input_Buffer(
                    Input_Buffer_Pointer ) :=
                      Hardware_Character_Buffer;

                  Input_Buffer_Pointer :=
                    Input_Buffer_Pointer mod
                      Max_Size_of_Internal_Input_Buffer + 1;

                  Buffer_Count := Buffer_Count + 1;
                end if;
              end Save_Hardware_Buffer_Character;
        end select;
      end loop;
end Interrupt_Input_Handler_Task;


task body Interrupt_Output_Handler_Task is

      Max_Size_of_Internal_Output_Buffer : constant POSITIVE
        := 64;

      Internal_Output_Buffer :
        array ( 1 .. Max_Size_of_Internal_Output_Buffer )
          of CHARACTER;

      Input_Buffer_Pointer  : POSITIVE := 1;
      Output_Buffer_Pointer : POSITIVE := 1:

      Buffer_Count : INTEGER := 1;

      Hardware_Character_Buffer : CHARACTER;
      for Hardware_Character_Buffer use at 16#0200#;

      Hardware_Character_Buffer_is_Empty : BOOLEAN := TRUE;

      Local_Output_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          MAC_Record_Type;
      Other_Task_MAC_Record :
        Mandatory_Access_Control_Types_Package.      --  see 3 - 3.1.3
          MAC_Record_Type;

    begin  -- Interrupt_Output_Handler_Task
      loop
        select
            accept Deposit_Character_into_Hardware_Buffer (
                    Other_Task_MAC_Record : in
                      Mandatory_Access_Control_Types_Package.
                        MAC_Record_Type ) do         --  see 3 - 3.1.3


              if Mandatory_Access_Control_Manager_Package.
                   Sensitivity_Labels_Match          --  see 3 - 3.1.3
                     ( Local_Output_Task_MAC_Record.
                         Sensitivity_Label,
                       Other_Task_MAC_Record.
                         Sensitivity_Label ) then

                Audit_Trail_Manager_Package.         --  see 3 - 3.2.3
                  Log_Subjects_Access_to_Object_in_Audit_Trail (
                    Local_Output_Task_MAC_Record,
                    Other_Task_MAC_Record );

                if Buffer_Count > 0 then

                  Hardware_Character_Buffer :=
                    Internal_Output_Buffer(
                      Output_Buffer_Pointer );

                  Output_Buffer_Pointer :=
                    Output_Buffer_Pointer mod
                      Max_Size_of_Internal_Output_Buffer + 1;

                  Buffer_Count := Buffer_Count - 1;

                else

                  Hardware_Character_Buffer_is_Empty := TRUE;
              end if;
            end if;
          end Deposit_Character_into_Hardware_Buffer;

        or
```

```
when Buffer_Count <
      Max_Size_of_Internal_Output_Buffer =>

  accept Put_Character_into_Interrupt_Output_Address
         ( Char                    : in CHARACTER;
           Other_Task_MAC_Record : in
             Mandatory_Access_Control_Types_Package.
               MAC_Record_Type ) do    --  see 3 - 3.1.3

    if Mandatory_Access_Control_Manager_Package.
         Sensitivity_Labels_Match          --  see 3 - 3.1.3
           ( Local_Output_Task_MAC_Record.
               Sensitivity_Label,
             Other_Task_MAC_Record.
               Sensitivity_Label ) then

      Audit_Trail_Manager_Package.         --  see 3 - 3.2.3
        Log_Subjects_Access_to_Object_in_Audit_Trail (
          Local_Output_Task_MAC_Record,
          Other_Task_MAC_Record );

      Internal_Output_Buffer(
        Input_Buffer_Pointer ) := Char;

      Input_Buffer_Pointer :=
        Input_Buffer_Pointer mod
          Max_Size_of_Internal_Output_Buffer + 1;

      Buffer_Count := Buffer_Count + 1;

      if Hardware_Character_Buffer_is_Empty then

        Hardware_Character_Buffer :=
          Internal_Output_Buffer(
            Output_Buffer_Pointer );

        Output_Buffer_Pointer :=
          Output_Buffer_Pointer mod
            Max_Size_of_Internal_Output_Buffer + 1;

        Buffer_Count := Buffer_Count - 1;

        Hardware_Character_Buffer_is_Empty := TRUE;
      end if;
    end if;
  end Put_Character_into_Interrupt_Output_Address;
```

```
            end select;
          end loop;
      end Interrupt_Output_Handler_Task;

   end Interrupt_Handler_Package;
```

2 - 13.9 Interface to Other Languages

Example: This example illustrates the interface of Ada with Pascal.

```
   package Graphics_Library_Package is
       procedure Draw_Circle
                    ( Center : in    Coordinates_Type;
                      Radius : in    Distance_Type );

       -- . . .

     private
       pragma INTERFACE ( PASCAL, Draw_Circle );

       -- . . .

   end Graphics_Library_Package;
```

2 - 14.7 Example of Input-Output

Example: This example illustrates trusted input and output.

```
    with TEXT_IO;
    with Mandatory_Access_Control_Types_Package;         -- see 3 - 3.1.3
    with Access_Control_List_Types_Package;
    generic

        Max_Characters_in_Message : POSITIVE := 80;

    package Generic_Sensitive_Text_File_Manager_Package is

      Sensitive_Text_File : TEXT_IO. FILE_TYPE;
      subtype Message_Type is STRING( 1 .. Max_Characters_in_Message );

      -- . . .
```

```
  procedure Put_Line (
    Subject_MAC_Record      : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Named_Object_MAC_Record : in
      Mandatory_Access_Control_Types_Package.
        MAC_Record_Type;

    Message                 : in Message_Type );

  -- . . .

end Generic_Sensitive_Text_File_Manager_Package;


with Mandatory_Access_Control_Manager_Package;       -- see 3 - 3.1.3
with Audit_Trail_Manager_Package;                    -- see 3 - 3.2.3
package Generic_Sensitive_Text_File_Manager_Package is

  -- . . .

  procedure Put_Line (
      Subject_MAC_Record      : in
        Mandatory_Access_Control_Types_Package.
          MAC_Record_Type;

      Named_Object_MAC_Record : in
        Mandatory_Access_Control_Types_Package.
          MAC_Record_Type;

      Message                 : in Message_Type ) is


    begin  -- Put_Line

      if Mandatory_Access_Control_Manager_Package.
           Sensitivity_Labels_Match                  -- see 3 - 3.1.3
             ( Subject_MAC_Record. Sensitivity_Label,
               Named_Object_MAC_Record. Sensitivity_Label ) then

        Audit_Trail_Manager_Package.                 -- see 3 - 3.2.3
          Log_Subjects_Access_to_Object_in_Audit_Trail (
            Subject_MAC_Record, Named_Object_MAC_Record );
            TEXT_IO. PUT_LINE ( Sensitive_Text_File, Message );

         end if;
     end Put_Line;

     -- . . .

  end Generic_Sensitive_Text_File_Manager_Package;
```

3.1.1 Discretionary Access Control

Example: This example illustrates a generic package specification for multiple instances of user identification and authentication (The package body is in Section 3 - 3.2.1.)

```
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
with Generic_Access_Control_List_Manager_Package;
generic

  -- . . .

  type Password_Type is private;

  -- . . .

package Generic_User_Identification_and_Authentication_Package is

    -- . . .

    procedure Check_Password
      ( Password          : in     Password_Type;

        Local_MAC_Record : in
          Mandatory_Access_Control_Types_Package.     -- see 3 - 3.1.3
            MAC_Record_Type;

        Password_is_Valid :   out BOOLEAN );

    -- . . .

end Generic_User_Identification_and_Authentication_Package;
```

3.1.3 Labels

Example: Sensitivity label type declaration

```
with Basic_TCB_Types_Package;
package Mandatory_Access_Control_Types_Package is

  -- . . .

  subtype Name_of_Resource_Type is
        Basic_TCB_Types_Package. Name_String_Type;
```

```
    subtype Sensitivity_Label_Type is
            Basic_TCB_Types_Package. Name_String_Type;

    Scrubbed_Sensitivity_Label : Sensitivity_Label_Type
      := Basic_TCB_Types_Package. Blank_Name_String;

    subtype Exception_Name_Type is
            Basic_TCB_Types_Package. Name_String_Type;

    Scrubbed_Exception_Name : Exception_Name_Type
      := Basic_TCB_Types_Package. Blank_Name_String;

    Others_String : Basic_TCB_Types_Package. Name_String_Type
      := Basic_TCB_Types_Package. Blank_Name_String;

    type MAC_Record_Type is
      record
        Name : Name_of_Resource_Type
          := Basic_TCB_Types_Package. Blank_Name_String; --  see 2 - 3.2.1

        Sensitivity_Label : Sensitivity_Label_Type
          := Scrubbed_Sensitivity_Label;

        -- . . .

      end record;

    -- . . .

  end Mandatory_Access_Control_Types_Package;


  with Basic_TCB_Types_Package;
  with Mandatory_Access_Control_Types_Package;
  package Mandatory_Access_Control_Manager_Package is

    -- . . .

    function Sensitivity_Labels_Match
      ( Sensitivity_Label_A : in
          Mandatory_Access_Control_Types_Package.
            Sensitivity_Label_Type;

        Sensitivity_Label_B : in
          Mandatory_Access_Control_Types_Package.
            Sensitivity_Label_Type )
        return BOOLEAN;

      -- . . .

    end Mandatory_Access_Control_Manager_Package;
```

3.2.1 Identification and Authentication

Example: This example illustrates a generic package body for multiple instances of user identification and authentication. (The package specification is in Section 3 - 3.1.1.)

```
    with Access_Control_List_Types_Package;
    with Audit_Trail_Manager_Package;                  -- see 3 - 3.2.3
    package body Generic_User_Identification_and_Authentication_Package is

        -- . . .

        procedure Check_Password
          ( Password          : in     Password_Type;

            Local_MAC_Record : in
              Mandatory_Access_Control_Types_Package.   -- see 3 - 3.1.3
                MAC_Record_Type;

            Password_is_Valid :   out BOOLEAN ) is

            -- . . .

          begin  -- Check_Password

            -- . . .

        end Check_Password;

        -- . . .

    end Generic_User_Identification_and_Authentication_Package;
```

3.2.3 Audit

Example: Audit trail manager package

```
with TEXT_IO;
with Mandatory_Access_Control_Types_Package;          -- see 3 - 3.1.3
package Audit_Trail_Manager_Package is

  -- . . .

  Audit_Trail_Text_File : TEXT_IO. FILE_TYPE;

  -- . . .

  procedure Log_Subjects_Access_to_Object_in_Audit_Trail (
    Subject_MAC_Record      : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type;

    Named_Object_MAC_Record : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type );

  -- . . .

  procedure Log_Exception_in_Audit_Trail (
    Exception_Name           : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        Exception_Name_Type;

    Exception_Raiser_Record  : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type;

    Exception_Handler_Record : in
      Mandatory_Access_Control_Types_Package.         -- see 3 - 3.1.3
        MAC_Record_Type );

  -- . . .

end Audit_Trail_Manager_Package;
```

```
with Mandatory_Access_Control_Manager_Package;
package body Audit_Trail_Manager_Package is

    -- . . .

    procedure Log_Subjects_Access_to_Object_in_Audit_Trail (
      Subject_MAC_Record       : in
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          MAC_Record_Type;

      Named_Object_MAC_Record : in
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          MAC_Record_Type ) is

        -- . . .

      begin  -- Log_Subjects_Access_to_Object_in_Audit_Trail

        -- . . .

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "Subject " & Subject_MAC_Record. Name );

        TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                            "  is not authorized to access " &
                             Named_Object_MAC_Record. Name );

        -- . . .

    end Log_Subjects_Access_to_Object_in_Audit_Trail;

    -- . . .

    procedure Log_Exception_in_Audit_Trail (
      Exception_Name             : in
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          Exception_Name_Type;

      Exception_Raiser_Record   : in
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          MAC_Record_Type;

      Exception_Handler_Record : in
        Mandatory_Access_Control_Types_Package.      -- see 3 - 3.1.3
          MAC_Record_Type ) is
      -- . . .

    begin  -- Log_Exception_in_Audit_Trail

      -- . . .

      if Mandatory_Access_Control_Manager_Package.
           Sensitivity_Labels_Match                    -- see 3 - 3.1.3
             ( Exception_Handler_Record.
                  Sensitivity_Label,
                 Exception_Raiser_Record.
                   Sensitivity_Label ) then

          TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                              "Subprogram " &
                              Exception_Handler_Record. Name &
                              " handled the exception, " );

          TEXT_IO. PUT_LINE ( Audit_Trail_Text_File, "  ", &
                              Exception_Name & " raised by " &
                              Exception_Raiser_Record. Name );

        else

          TEXT_IO. PUT_LINE ( Audit_Trail_Text_File,
                              "Subprogram " &
                              Exception_Handler_Record. Name &
                    " is not authorized to handle the exception, " );

          TEXT_IO. PUT_LINE ( Audit_Trail_Text_File, "  ", &
                              Exception_Name & " raised by " &
                              Exception_Raiser_Record. Name );
      end if;

      -- . . .

    end Log_Exception_in_Audit_Trail;

    -- . . .

  begin  -- Audit_Trail_Manager_Package initialization

    Mandatory_Access_Control_Types_Package.            -- see 3 - 3.1.3
      Others_String( 1 .. 6 ) := "others";

end Audit_Trail_Manager_Package;
```

6.0 BIBLIOGRAPHY

Abrams, Marshall D., Podell, Harold J., 1987. Tutorial Computer and Network Security. Washington, D. C.: IEEE Computer Science Press.

Abrams, Marshall D., Podell, Harold J., 1988. Recent Developments in Network Security. 2906 Covington Road, Silver Spring, MD, 20910.: Computer Educators Inc.

Anderson, Eric R. "Ada's Suitability for Trusted Computer Systems" from Proceedings of the Symposium on Security and Privacy, Oakland, California, 22-24 April, 1985.

Baker, T. P. 13 July 1988. Issues Involved in Developing Real-Time Ada Systems. Department of Computer Science, Florida State University, Tallahasse, FL: for U. S. Army HQ, COMM/ADP.

Boebert, W. E., Kain, R. Y., and Young, W. D., July 1985. "Secure Computing: The Secure Ada Target Approach." Scientific Honeyweller, Vol. 6, No. 2.

Booch, Grady. 1987A. Software Components with Ada. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Booch, Grady. 1987B. Software Engineering with Ada. 2nd ed. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Brill, Alan E., 1983. Building Controls Into Structured Systems. New York, N. Y.: YOURDON Press Inc.

Buhr, R. J. A., 1984. System Design with Ada. Englewood Cliffs, N. J.: Prentice-Hall.

Cherry, George W., 1984. Parallel Programming in ANSI Standard Ada. Reston, Virginia: Reston Publishing Company, Inc.

Feldman, Michael B. 1985. Data Structures with Ada. Reston, Virginia: Reston Publishing Company, Inc.

Final Evaluation Report of SCOMP 23 September 1985. Secure Communications Processor STOP Release 2.1.

Freeman, Peter. 1987. Tutorial: Software Reusability. Washington, D. C.: IEEE Computer Science Press.

Gasser, Morrie 1988. Building a Secure Computer System. New York, N. Y.: Van Nostrand Reinhold Company, Inc.

Gehani, Narain. 1984. Ada Concurrent Programming. Englewood Cliffs, N. J.: Prentice-Hall Inc.

Gilpin, Geoff. 1986. Ada: A Guided Tour and Tutorial. New York, N. Y.: Prentice Hall Press.

Goodenough, John B., "Exception Handling: Issues and a Proposed Notation," Communications of the ACM, 18(12):683-696, December 1975.

Hadley, Sara, Hellwig, Frank G. of the National Security Agency, and Rowe, Kenneth, CDR Vaurio, David of the National Computer Security Center. 1988. "A Secure SDS Software Library," Proceedings, 11th National Computer Security Conference, Baltimore, MD, October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.

IEEE Standard Glossary of Software Engineering Terminology. 18 February 1983. (IEEE Std 729-1983).

Luckham, David C., von Henke, Friendrich W., Krieg-Brueckner, Bernd, Owe, Olaf, "ANNA-A Language for Annotating Ada Programs, Preliminary Reference Manual," Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

Mungle, Jerry. 1988. Developing Ada Systems. Technology Training Corporation's seminar.

National Computer Security Center. 1985. Department of Defense Trusted Computer System Evaluation Criteria. (DOD 5200.28-STD)

National Computer Security Center. 1987. Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria.

Nissen, John and Wallis, Peter. 1984. Portability and Style in Ada. Cambridge, Great Britain: Cambridge University Press.

Odyssey Research Associates, Inc., Toward Ada Verification, Preliminary Report" (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, March 25, 1985.

Reference Manual for the Ada Programming Language. 1983. ANSI/MIL-STD-1815A-1983, 17 February 1983.

Ross, D. T., Goodenough, J. B., and Irvine, C. A., 1975. "Software Engineering: Process, Principles, and Goals," Computer.

Saydjari, O. S., Beckman, J. M., and Leaman, J. R. 1987. "LOCKing Computers Securely," Proceedings, 10th National Computer Security Conference, Baltimore, MD, September 21-24, 1987, National Bureau of Standards/National Computer Security Center.

Shaffer, Mark of Honeywell, Computing Technology Center, and Walsh, Geoff of R & D Associates Secure. 1988. "LOCK/ix: On Implementing Unix on the LOCK TCB," Proceedings, 11th National Computer Security Conference, Baltimore, MD, October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.

Tracz, Will. 1988. Tutorial: Software Reuse: Emerging Technology. Washington, D. C.: IEEE Computer Science Press.

Tripathi, Anand R., Young, William D., Good, Donald I., "A Preliminary Evaluation of Verifiability in Ada," Proceedings of the ACM National Conference, Nashville, TN, October 1980.

Trusted Computer System Security Requirements Guide for DoD Applications. 1 September 1987.

APPENDIX D

Abstracts of Three
Frequently Referenced
Documents

Prepared for:

National Computer Security Center
9800 Savage Road
Fort Meade, MD 10755

Prepared by:

Ada Applications and
Software Technology Group

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

April 1989

This appendix contains an abstract for each of three documents referenced by this report concerning the LOCK hardware technology. Where available, the abstract was taken directly from the referenced document; in other instances, an abstract was either written for a document or the existing abstract was expanded.

SECURE COMPUTING: THE SECURE ADA TARGET APPROACH

The paper proposes to secure computing with a small trustworthy hardware subsystem of a computer that is not subject to the potential penetrations inherent in software. This hardware subsystem "enforces strict rules that protect data from hostile users. The subsystem can be proved trustworthy in a more rigorous sense than can earlier mechanisms." This subsystem functions as the reference-monitor mechanism, or security kernel. Overall, this subsystem "consists of three state spaces, or collections of three state spaces, or collections of stored information. The state spaces are called the value state, the protection state and the underlying abstractions. The value state consists of all objects that can possibly be made visible to subjects; it is therefore outside the reference monitor. The protection state and the underlying abstractions are the information that the reference monitor keeps internal to itself and uses in making access decisions. At this level the protection state is represented as a matrix and the underlying abstractions are represented as tables. The paper introduces "the concept of an operation invoked by a subject and performed by the reference monitor. This operation consults the underlying abstractions and updates the protection state accordingly. The paper further details the design and operation of this hardware reference monitor as applied to creating a tamper resistant Ada target.

LOCKING COMPUTERS SECURELY

Progress has been slow over the last 15 years in the relatively new field of computer security. Every initiative started from scratch to develop a secure computer. First prototypes, built in software, were slow and difficult to use. LOCK is a technology research and development project to build a hardware-based Reference Monitor module. This module will be generic and thus reusable on many different computers. Full advantage will be taken of inexpensive generic cryptographic modules currently in development.

LOCK/ix: On Implementing Unix on the LOCK TCB

In the LOCK/ix version, the LOgical Coprocessing Kernel (LOCK) is a Trusted Computing Base (TCB) that is designed to meet and exceed the requirements for a Class A1 secure system. This paper describes the results of a study that determined how to port the Unix System V Operating System to the LOCK TCB, while maintaining maximum compatibility with the System V Interface Definition (SVID) [SVID86].

BIBLIOGRAPHY

Boebert, W. E., Kain, R. Y., and Young, W.D. July 1985. "Secure Computing: The Secure Ada Target Approach." Scientific Honeyweller, Vol. 6, No. 2.

Saydjari, O. S., Beckman, J. M., and Leaman, J. R. 1987. "LOCKing Computers Securely," Proceedings, 10th National Computer Security Conference, Baltimore, MD, September 21-24, 1987, National Bureau of Standards/National Computer Security Center.

Shaffer, Mark of Honeywell, Computing Technology Center, and Walsh, Geoff of R & D Associates Secure. 1988. "LOCK/ix: On Implementing Unix on the LOCK TCB," Proceedings, 11th National Computer Security Conference, Baltimore, MD, October 17-20, 1987, National Institute of Standards and Technology/National Computer Security Center.